DOMAIN MODELING OF TIME-DEPENDENT SYSTEMS

THESIS

Robert W. Waggoner
Captain, USAF

AFIT/GCS/ENG/93D-23

AFIT/GCS/ENG/93D-23

DOMAIN MODELING OF TIME-DEPENDENT SYSTEMS

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Robert W. Waggoner, B.S.

Captain, USAF

December, 1993

Acknowledgements

I would like to thank my thesis advisor, Dr. Thomas C. Hartrum, for his many contributions to this research – for the hours of thought provoking discussions, for providing guidance when needed, for allowing me the freedom to tailor the research into areas of personal interest, and for his expert commentary on the technical content and format of this document.

Thanks also to committee members Major Paul Bailor and Major David Luginbuhl for their insightful comments and suggestions.

Captains Randour and Anderson, the original creators of Architect, also deserve credit for providing a solid foundation upon which this research builds.

Dan Zambon and David Doak did a fantastic job keeping the computer network up and running. I'd [illegible]e to thank them for allowing me to use *their* laboratory!

Many th[illegible] to the other members of the KBSE research group, especially Captains Welgan and Gool whose work interleaved with mine. And thanks to adopted KBSE member Captain Jeff Miller for providing the humorous sound bytes that helped preserve sanity when things got crazy.

Most of all, I'd like to thank my family for their support and encouragement, and for understanding why I couldn't always spend as much time with them as they deserved. This thesis is dedicated to them.

Robert W. Waggoner

Table of Contents

Page

Acknowledgements ii

List of Figures vii

List of Tables viii

Abstract ix

I. Introduction 1

1.1 Background 1

1.2 Problem Description 3

1.3 Scope 3

1.4 Approach 3

1.5 Sequence of Presentation 4

II. Literature Review 5

2.1 Introduction 5

2.2 Domain Analysis 5

2.3 Software Architectures 8

2.3.1 VHDL 8

2.3.2 OCU 9

2.3.3 Architect 11

2.4 Simulations 12

2.4.1 Discrete versus Continuous Systems 12

2.4.2 Event-Driven and Time-Driven Simulations 13

2.5 Conclusion 13

Page

III. Domain Analysis of the Time-Dependent Logic Circuits Domain 14
3.1 Introduction 14
3.2 Stage 1: Defining the Scope of the Domain 14
3.2.1 Logic Circuits Domain 14
3.2.2 The Domain of Interest 15
3.2.3 Block Diagram 16
3.2.4 User Needs 16
3.3 Stage 2: Define/refine domain-specific concepts/requirements . . 17
3.3.1 Object Model 18
3.3.2 Concept of Operation 20
3.3.3 Stale Events 22
3.3.4 Dynamic Models 23
3.3.5 Functional Models 26
3.4 Stage 3: Define/refine domain-specific design and implementation constraints 26
3.5 Stage 4: Develop domain architectures/models 27
3.5.1 Sensitivity 28
3.5.2 Domain-Specific Semantic Checks 29
3.5.3 Architectural Syntax and Semantic Checks 29
3.5.4 Implementation of Domain-Specific Semantic Checks . . 30
3.6 Stage 5: Produce/gather reusable workproducts 30
3.6.1 Object Functions 31
3.6.2 Object-Class Hierarchies in the Logic Circuits Domain . 32
3.6.3 Domain-Specific Language 33
3.6.4 Test Methodology 34
3.6.5 Test Results 34
3.7 Summary 38

Page

IV. Design and Implementation of the Cruise Missile Technology Base 40

4.1 Introduction 40

4.2 Stage 1: Define the Scope of the Domain 40

4.2.1 Scoping the Domain 40

4.2.2 User Needs 41

4.3 Stage 2: Define/refine domain-specific requirements/concepts. . . 42

4.3.1 System Requirements and Concepts 42

4.3.2 Object-Oriented Analysis 43

4.3.3 Propulsion Subsystem 43

4.3.4 Airframe Subsystem 47

4.3.5 Avionics Subsystem 48

4.3.6 AutoPilot 54

4.3.7 Warhead Subsystem 57

4.4 Stage 3: Define/refine domain-specific design and implementation constraints. 58

4.5 Stage 4: Develop domain architectures/models 58

4.6 Stage 5: Produce/gather reusable workproducts 60

4.6.1 Development 60

4.6.2 Validation 61

4.6.3 Performance Considerations 65

4.7 Summary 67

V. Conclusions and Recommendations 68

5.1 Introduction 68

5.2 Research Goals 68

5.3 Results 68

5.3.1 Impact on the Architecture and Executive 68

5.3.2 Architect's New Capabilities 69

Page

5.4 Evaluation of the Domain Engineering Process 70

5.5 Suggestions for Improvements 71

5.5.1 Event Ordering 71

5.5.2 Architectural Syntax and Semantic Checks 72

5.5.3 Visualization Support 72

5.6 Suggestions for Further Research 72

5.7 Conclusion 74

Appendix A. REFINE Code Listings for Architect 75

Vita 76

Bibliography 77

List of Figures

Figure		Page
1.	VHDL Model of Time	9
2.	Object Connection Update (OCU) Model	10
3.	A Sequentially Executing Application in Architect	11
4.	Block Diagram of Event-Driven Domain in Architect	19
5.	Simplified Functional Moael	21
6.	General Dynamic Model for Objects Without Internal State	23
7.	Dynamic Model for the Counter Object	24
8.	Dynamic Model for the Switch Object	24
9.	Dynamic Model for the JK-Flip-Flop Object	25
10.	Dynamic Model for the One-Shot Object	26
11.	Dynamic Model for the Clock Object	27
12.	Modified OCU Subsystem with Events	28
13.	Counter Test	35
14.	Simple Oscillator	36
15.	Elements of the Cruise Missile Domain	41
16.	Object Model of the Cruise Missile Domain of Interest	44
17.	Dynamic and Functional Flow Diagrams for the Fuel-Tank	45
18.	Functional Flow Diagram for the Throttle	46
19.	Dynamic and Functional Flow Diagrams for the Jet-Engine	47
20.	Functional Flow Diagram for the Airframe	49
21.	Dynamic and Functional Flow Diagrams for Navigation	51
22.	Dynamic and Functional Flow Diagrams for Guidance	53
23.	Functional Model for AutoPilot	55
24.	Functional Model of the Cruise Missile	59
25.	Closed Course Flight Path Performance	64
26.	Missile Airspeed	66

List of Tables

Table		Page
1.	Truth Table - 3-to-8 Line Decoder	16
2.	Truth Table - 4-Input Multiplexer	16
3.	Truth Table - JK FLIP-FLOP	17
4.	Truth Table - Half Adder	17
5.	Comparison of JK Flip-Flop Attributes	38
6.	Fuel-Tank	43
7.	Throttle	45
8.	Jet-Engine	46
9.	Airframe	48
10.	Navigation	50
11.	Guidance	52
12.	AutoPilot	54
13.	Warhead	57

AFIT/GCS/ENG/93D-23

Abstract

This research investigated the feasibility of composing time-dependent specifications in Architect, a domain-oriented application composition and generation system being developed at the Air Force Institute of Technology (AFIT). Architect composes formally specified domain objects into an executable software specification that can be used to verify program correctness prior to generation of language specific code. As part of this research, domain modeling techniques were investigated and a candidate process was selected for evaluation. The process was used to develop domain models for two diverse time-dependent domains. Using object-oriented analysis, formal specifications were developed for a collection of event-driven logic circuit components and a collection of time-driven cruise missile components. Applications from each domain were composed in Architect and executed to verify correct behavior.

DOMAIN MODELING OF TIME-DEPENDENT SYSTEMS

I. Introduction

1.1 Background

Much of the imprecision in software specifications occurs during the transfer of knowledge about the problem space from the application specialist who defines the problem to the software engineer who implements a solution. The application specialist and the software engineer are each fluent in the language of his own domain but are often unversed in the language of the other. Because there is no common formal language with which to communicate requirements, the transfer of knowledge is generally done through informal specifications or in a natural language such as English. Imprecise specifications frequently result in software solutions that do not meet user expectations or requirements.

A solution currently being investigated by the Knowledge Based Software Engineering (KBSE) Research Group at the Air Force Institute of Technology (AFIT) involves development of domain-oriented tools that allow the application specialist to formally specify requirements within his own domain. Working in a familiar environment using domain-specific terminology (textual interface) and/or symbology (graphical interface), the application specialist can develop his own applications without need for an intermediate mapping to an informal specification. If the formal specifications are mapped to an executable specification language, the application can be tested for proper behavior and functionality, and modified as required until proper behavior is exhibited. The KBSE research group at AFIT is developing a domain-oriented application composition system called Architect. The original version, developed in 1992 by Randour (18) and Anderson (1), had a textual interface. In 1993, Weide (24) developed a graphical interface called Architect Visual System Interface (AVSI). Architect is built within the Software Refinery environment which consists of the Refine^{TM} wide-spectrum language, the Dialect grammar tool, and the Intervista graphical tool (20). As a wide-spectrum language, Refine is suitable for expressing applications at the code level or at the specification level. At the code level, the user specifies *how* transformations are implemented; at the specification level, the user only has to specify *what* transformations are needed. Using high-level abstractions such as set formers, rules, and transforms, the user can specify a set of post-conditions that must be made true whenever a set of pre-conditions are true and not be concerned with the mechanisms of how the post-conditions are met.

A knowledge base in Architect consists of domain objects, a domain-specific language (DSL), and icon descriptions. Each domain object (class) contains a Refine code description of its interfaces, attributes, and functions. The DSL is a Dialect grammar that defines the format for saving and parsing text file descriptions of domain objects. Icon descriptions define how each object class appears in the graphical interface and are required only when applications are composed with AVSI.

Once a knowledge base is validated, the application specialist has a library of components with known properties that he can use to create applications. The domain knowledge that is encapsulated in the domain objects is reused each time a new application is composed.

The reuse of knowledge is new to software engineering, but it is a fundamental concept in other engineering disciplines. Traditional engineering disciplines use models to capture knowledge about a domain. When a new project is undertaken, engineers use the models to assist in the design process. The models are instantiated with design parameters and the resulting behavior analyzed. Parameters can be adjusted until desired behavior is attained. If no suitable models exist, new ones are developed and added to the knowledge base, creating a growing body of knowledge. Without libraries of reusable software components to draw upon, new software must be created ad-hoc for each project. This results in costly, error-ridden software that does not meet user needs. Model-based software development is an attempt to apply traditional engineering practices to software — to put *engineering* into software engineering (5).

1.2 Problem Description

Prior to this research, the only technology base available for Architect was a set of 12 logic circuit primitives. At the time this technology base was developed, Architect had no executive. Instead, a rudimentary execution capability was provided that required the application specialist to specify a fixed sequence of execution for primitives in the application. The logic circuits domain provided insight into some of the limitations of Architect. The fixed sequence of execution restricted the types of primitive behaviors that could be modeled and limited the complexity of applications that could be composed. Specifically, time-dependent primitive behavior and compositions with feedback were beyond Architect's capabilities. More experience with new and diverse domains was needed to determine the requirements for a composition system that supports a broad spectrum of domains.

Problem Statement: To demonstrate the feasibility of composing time-dependent specifications using the OCU/Architect architecture; to extend the Architect technology base by developing two diverse domain models for time-dependent domains.

1.3 Scope

This research focuses on developing domain models for two diverse time-dependent domains. The first domain is an extension of the original logic circuits domain, which is not time-dependent. The second domain involves a moving vehicle, specifically, a cruise missile.

1.4 Approach

The following steps were taken to accomplish the objectives of this research:

1. The first step was to gain an understanding of domain analysis. Several articles in the literature discuss domain analysis from a reuse viewpoint, and a few suggest methodologies for performing domain analysis. Being a rather new field of study, the requirements of domain analysis are still being defined; consequently, its processes are still evolving. A process developed by Tracz, Coglianese, and Young at IBM Federal

Sector Company for use on domain-specific software architectures (23) contained several features applicable to this research and was selected to be used as a guide for the domain analyses performed in this research.

2. The next step was to perform a domain analysis on the time-dependent logic circuits domain using the techniques expounded in Tracz' process. A small but representative set of logic circuits was selected and an object-oriented analysis (OOA) was performed to determine the significant properties and behaviors of each device. From these models, architectural constraints and executive requirements were identified that required modifications to the existing software architecture in Architect. The architectural modifications and executive development were performed by other researchers (6) (25). The new circuits were then implemented in Refine code and tested. Finally, sample applications were composed and tested to verify proper interaction of the components.

3. Next, a domain analysis was performed on the cruise missile domain using Tracz' methodology. The steps involved essentially paralleled those described for the circuits domain except that no new architectural requirements were identified.

1.5 *Sequence of Presentation*

The next chapter reviews the current literature applicable to this thesis. Chapter III discusses the domain analysis of the time-dependent logic circuits domain and presents the results. Chapter IV discusses the domain analysis of the cruise missile domain. Chapter V presents conclusions about this research effort and makes recommendations for further research in this area.

II. Literature Review

2.1 Introduction

The primary objective of the literature review was to examine current research in the area of domain analysis to support development of the two domain models. Other systems similar to Architect and relevant software architectures were also examined.

2.2 Domain Analysis

Prieto-Díaz defines domain analysis as "a process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems" (16:47). The process' output is a domain model which describes the objects, operations and relationships common to all systems in the domain. He notes that domain analysis efforts are more likely to produce quality results when several ad-hoc applications have previously been developed in the domain. The ad-hoc applications contribute to the understanding of the domain. When the domain is sufficiently well understood, its salient features can be abstracted and codified into a domain model. In Prieto-Díaz' view, the domain model is a domain-specific language containing the syntax and semantic rules that represent all objects and operations in the domain. Don Batory describes a domain model as "a theory of how software systems of a domain can be constructed from prefabricated components" (2). Domain modeling, therefore, is the process of identifying those domain components and determining their attributes, functions, and relationships.

In 1987, Prieto-Díaz called domain analysis a "knowledge intensive activity for which no methodology or any kind of formalization is yet available (17:63)". He saw prior domain analysis attempts as being *product-oriented* rather than *process-oriented*. In order to help formalize the domain analysis process, he proposed a methodology consisting of eight steps. The first three steps are precursors to the actual analysis and provide the framework in which the analysis will be performed. The final two steps, performed after domain analysis is completed, relate to implementation and reuse.

1. Define and scope the domain - determine the boundary of the domain to limit the type and quantity of information to be analyzed.
2. Identify sources of knowledge and information about the domain - locate domain expertise and documentation about existing systems
3. Define the approach - establish guidelines for extracting relevant knowledge from the domain expert and and from existing systems.
4. Identification of objects and operations - determine common characteristics of similar systems in the domain and generalize into a set of objects and operations on the objects.
5. Abstraction - determine the relationships between objects.
6. Classification - group objects sharing common attributes or properties to create a taxonomy of the domain; create a DSL that .an be used to describe the objects and operations in the domain.
7. Encapsulation - make selected components reusable through modularization, structured design, and standardization of interfaces.
8. Produce reusability guidelines - develop documentation for reusers.

Tracz, Coglianese, and Young, working on domain-specific software architectures (DSSA), believed that existing domain analysis processes focused too much on the the solution space instead of the problem space because they fail to distinguish types of requirements. Tracz' approach to domain analysis differentiates requirements as either *functional requirements* or *implementation constraints* where the former define the problem space and the latter characterize the solution space. They developed a five stage *domain engineering process* to "map user need into system and software requirements that, based on a set of implementation constraints, define a DSSA"(23:1).

Each of the five stages of their process consists of a series of substages. The stages define a "concurrent, recursive, and iterative" process; therefore, each stage may be revisited several times as the domain model develops in order to correct oversights, incorporate new requirements, or add refinements (23:2). The stages of the process are outlined below:

1. *Define the scope of the domain.* The purpose of this stage is to bound the domain of interest and to determine the goals with primary emphasis on determining/meeting the user's needs. Consultations with domain experts, reviews of relevant research or documentation, and analysis of existing systems support this effort. The main

outputs of this stage are a high-level block diagram of the domain and a list of users' needs.

2. *Define/refine domain-specific concepts/requirements.* Using the products of the previous stage as inputs, this stage identifies the entities (objects) in the domain along with their attributes, data flows, control flows, and relationships to other domain entities. Entities are classified and common entities are grouped. A dictionary with domain-specific terminology and a high-level requirements specification are developed. The primary output is an object-oriented analysis of the domain.

3. *Define/refine domain-specific design and implementation constraints.* The objective of this stage is to identify the constraints imposed upon the architecture by the performance requirements of applications in the domain. Constraints such as "how fast", "how often", and "how big" are identified and their impact on the design and implementation is determined.

4. *Develop domain architectures/models.* The goal of this stage is to determine the generic architectures needed to implement the range of applications in the domain. Hierarchical decomposition of applications into subsystems, lower level subsystems, and leaf modules is performed. Subsystem or module requirements such as concurrent versus sequential execution and massively parallel versus single processor host environments will drive a need for multiple architectures to support domain applications. This stage also produces the syntactic and semantic rules for composing objects. The main outputs from this stage are the architectures with component interfaces, and mappings between the subsystem/modules and the Stage 2 requirements.

5. *Produce/gather reusable workproducts.* In this last stage, the domain objects are implemented (coded) as reusable components that can be composed to create new domain applications. Commercial off-the-shelf (COTS) and existing in-house components are evaluated for suitability and modified if practical. Otherwise, new components are developed. The outputs of this stage are the reusable components, their test cases, and the documentation needed to use the components. Output also includes a cross reference of functional requirements and constraints to specific components.

Tracz' domain engineering process provided the framework for the domain analyses performed in this research.

2.3 Software Architectures

The software architecture provides the infrastructure that allows domain objects to be composed into applications. It enforces domain-independent syntax and semantics rules about how objects can be connected and provides the structural form of the application, e.g., how the domain objects are grouped into subsystems. The software architecture also provides the mechanisms whereby objects communicate with other objects or subsystems and provides the interface to the application executive.

2.3.1 VHDL. VHDL is a hardware description language used to describe digital hardware devices. A VHDL description of a device involves a behavioral model, a timing model, and a structural model. The behavioral model views devices as having a set of processes which transform their inputs into outputs. All processes in the model are considered to be concurrent.

VHDL has a two-stage model of time based upon a stimulus-response paradigm as depicted in Figure 2.3.1. During the first stage, signals are propagated to the devices' inputs (the stimulus). During the second stage, device processes are executed to update output values (the response) (14). VHDL entities only respond to inputs they are "sensitive" to as defined by *sensitivity channels* (14:10). Thus a clocked device might be defined to only be sensitive to transitions on its clock input. A transition on the device's clock input would cause the device's process(es) to be executed, but transitions on other inputs would not.

The temporal behavior of VHDL devices is modeled by three types of delays (14:75-82), (14:13):

1. *Transport delay* - Time from when a process calculates a new output value until the new value is made available to other devices. Analogous to propagation delay through a device or wire.
2. *Inertial delay* - Amount of time a signal must be stable before being considered valid. Used to simulate rejection of noise or transients on an input line.

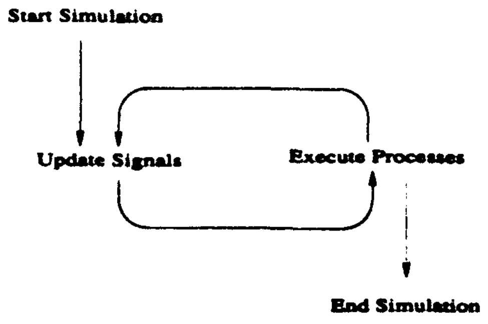


Figure 1. VHDL Model of Time

3. *Delta delay* - The zero time delay between stages of the simulation cycle. Data calculated by a process is not available to other devices until the start of the next simulation cycle. Used when no transport delay is given; does not cause simulation clock to change.

The structural model describes how devices are grouped into new units called subsystems.

2.3.2 OCU. The Object Connection Update (OCU) model is a software architecture developed by the Software Engineering Institute (SEI) (11). The main element in the OCU model is the subsystem. An OCU subsystem consists of an import area, an export area, and a controller. The controller controls one or more controllees which may be objects or other subsystems. All data inputs required by the subsystem can be found in the import area. Similarly, all subsystem outputs required by other objects or subsystems are placed in the export area. An important aspect of the OCU model is anonymity. Objects have no knowledge of other objects or of subsystems. Subsystems know only about their controllees, and have no knowledge of other subsystems (11). In the SEI concept, subsystems are the "locus of the mission" while objects are the "services to carry out the mission"(11:18). Figure 2 depicts a subsystem in the OCU architecture. Two other elements in the OCU model are the executive and surrogates. The executive controls the top-level subsystems; the surrogates provide I/O services to/from the host environment.

Each subsystem has a common set of functions which are called to invoke particular actions. These functions are listed below:

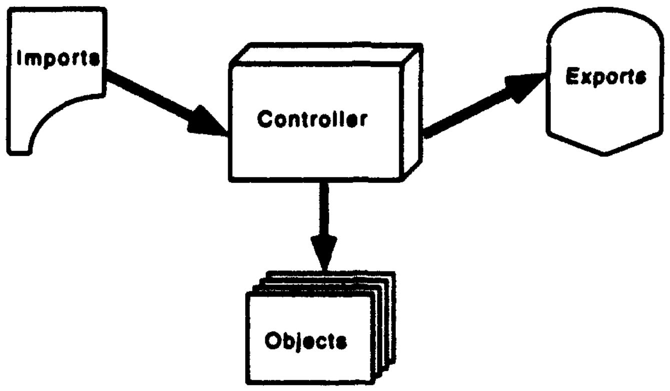


Figure 2. Object Connection Update (OCU) Model

- Update - Causes the controller to update the state of the subsystem based upon current state and data in the import area; state data needed externally is written to the export area. Subsystems have no explicit state; it is derived from the aggregation of its controllees' states; consequently, subsystems have no attributes.
- Stabilize - Causes the subsystem to reach a state of equilibrium consistent with its import data and current state.
- Initialize - Creates objects (controllees) and sets their initial states.
- Configure - Maps controllee inputs to the import area and controllee outputs to the export area. Defines how the subsystem is "connected" internally and externally.
- Destroy - Deallocates objects created during initialization.

Similarly, there is a common set of functions for each object:

- Update - Causes the object to calculate a new state based upon its inputs and current state.
- SetState - Used to change an object's state directly.
- Create - Instantiates a new object.
- SetFunction - Alters the algorithm used by the Update function.
- Destroy - Deallocates the object.

Objects can have the following types of attributes:

- Constants - Attributes whose values may not be changed; the OCU model does not specify whether these are absolute constants such as π or configurable constants whose values do not change during execution.
- Coefficients - Attributes used in the update function
- State Variables - Attributes that determine the state of the object

2.3.3 Architect. Architect is based on the OCU model. Existing applications in Architect execute in a non-event-driven sequential mode. Each subsystem contains an update algorithm that determines the order in which the subsystem's objects are updated. The update algorithm is designed to use IF-THEN-ELSE statements and WHILE loops to control the sequencing of object updates; however, not all required functionality has been implemented to take advantage of these conditional constructs.

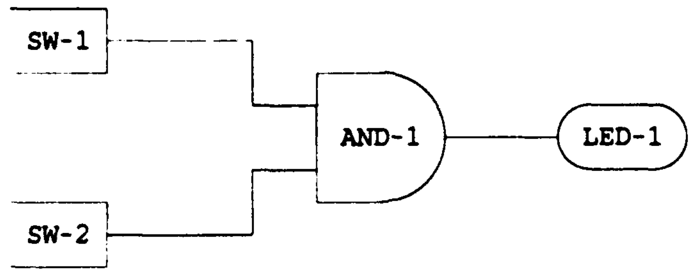


Figure 3. A Sequentially Executing Application in Architect

Figure 3 shows a simple application containing two switches (SW-1 and SW-2), an and-gate (AND-1), and a Light Emitting Diode (LED-1). The update algorithm for a subsystem containing these objects would be:

- update SW-1
- update SW-2
- update AND-1
- update LED-1

Prior to being updated, there is no data in the objects' output areas. The application specialist must select the sequence in such a way that no object is updated before all other objects providing its input data are updated. In this example, the order of the switch updates could be reversed, but any other sequence would be erroneous. Applications constructed this way do not allow feedback since feedback would require the availability of data that has not yet been generated.

Create and *destroy* are implemented in Architect as static operations. OCU elements are created during the composition process and exist as persistent objects in the Refine object base; they are deleted after execution through an erase function. Dynamic allocation and deallocation are possible in Refine but no attempt has been made to incorporate them into Architect. The current technology bases have no requirement for dynamic allocation/deallocation, but this could change when more complex domains are added to the technology base. The main impediment is not creating and destroying the objects, but rather dynamically configuring the objects' connections and modifying the controllers' update algorithms. Dynamic allocation/deallocation is not investigated as part of this research. Architect has a domain-independent SetState function that can be used to modify object attributes directly in the object base. The function has not been implemented as a domain object function.

2.4 Simulations

2.4.1 Discrete versus Continuous Systems. A system can be classified as discrete or continuous according to the way system state changes with respect to time. System state is determined by the values of all the state variables in the system. A discrete system is one in which all state variables have discrete values. The state of a discrete system changes in finite jumps or *quanta* according to the discrete values of its state variables. State transitions are caused by stimuli called *events*. Events are associated with an instantaneous point in time and have no duration. A discrete system can be modeled as a finite state machine. A continuous system has state variables that can take on a continuous range of values. The state of a continuous system is characterized by smooth, continuous changes. Continuous systems are often represented by sets of differential equations that specify the system's behavior. A *hybrid* system is one that has both discrete and continuous variables (7:3).

Consider a system consisting of cars passing through a busy intersection during rush hour. The number of cars waiting at any time for the traffic light to change is an example of a discrete variable since it can only take on integral values. In contrast, the average speed of the vehicles passing through the intersection is a continuous variable since speed can take on a continuous range of values. If the speed need only be known to the nearest mile per hour, then it could also be modeled as a discrete variable.

Continuous variables cannot be measured or represented with infinite precision. The resolution of the measuring device limits how accurately the variable can be measured, and the number of bits used by the computer to store the variable's value limits the precision to which the value can be represented. A continuous variable is ultimately represented as a discrete variable due to one of these limiting factors. Discrete variables can also be modeled as continuous variables with a sufficiently fine granularity of representation, such as using a fixed point number to represent an integer value. The choice to model systems as discrete or continuous is one of practicality. The logic circuits domain is more easily modeled as discrete; the cruise missile domain is more easily modeled as continuous.

2.4.2 Event-Driven and Time-Driven Simulations. An important characteristic of simulations is the way in which the advancement of time is handled. In an event-driven simulation, events are raised asynchronously by components in the simulation. As the events are processed, the simulation clock is updated so that its time is the same as the time of the event currently being processed. The clock is, therefore, driven by the execution of the simulation model. In a time-driven (or time-stepping) simulation, the executive advances the clock in uniform increments. At each increment, model components are given the opportunity to execute. The model components respond to changes in the clock but do not raise new events. After each component has been given the opportunity to execute, the executive increments the clock and repeats the process.

2.5 Conclusion

The current literature provides no definitive approach to domain analysis; the processes which produce successful domain models are still evolving. Prieto-Díaz, and Tracz have identified processes that a domain analyst can use as guides. Although none of these processes can be considered algorithmic, they did prove useful to the domain analyses for this research. In addition, VHDL provided valuable insight into the requirements for implementing the time-dependent logic circuits domain objects and their interaction with the architecture and executive.

III. Domain Analysis of the Time-Dependent Logic Circuits Domain

3.1 Introduction

This chapter discusses the domain analysis performed on the time-dependent logic circuits domain in the context of Tracz' domain engineering process. The chapter is organized around the five stages of the process.

3.2 Stage 1: Defining the Scope of the Domain

Defining the scope of the domain is the first stage in Tracz' process. According to Tracz, the emphasis in this stage should be to accurately define the users' needs (23:2). The main outputs of this stage are a list of user needs and a block diagram of the domain showing inputs, outputs, and high-level relationships.

3.2.1 Logic Circuits Domain. The realm of logic circuits is very broad. At the lower end of the spectrum are the small-scale integration (SSI) devices, containing up to about 12 gates. Slightly more complex devices containing up to 100 gates are classified as medium-scale integration (MSI). Counters, shift-registers, and decoders are typical devices in this category. Large-scale integration (LSI) devices, containing up to 1000 gates, are used to implement more complex devices such as small memories or programmable logic arrays. At the upper end of the spectrum are the very-large-scale integration (VLSI) devices which may contain several hundred thousand gates. Large memories and CPUs are examples of VLSI devices (15:169, 297). Within a category, there may be families of devices with similar properties. For example, transistor-transistor-logic (TTL), emitter-coupled-logic (ECL), and complementary-metal-oxide-semiconductor (CMOS) are common families within the SSI and MSI technologies. Each family has characteristic speed, power dissipation, threshold voltages (input voltages representing logic levels), fan-out (the maximum number of inputs that an output can drive), etc. Devices within the same family have compatible inputs and outputs; thus, they can be composed together directly. Devices from different families generally require interface support to make their inputs and outputs compatible. Almost all devices operate on binary input data with two discrete logic levels represented by high/low, true/false, asserted/negated, or one/zero. Many devices have a tri-state output that can either be at one of the two logic levels or in a high-impedance state which effectively "disconnects" the output from other devices. The tri-state output makes it possible to connect the output and other similar outputs to a common bus and to only allow one device at a time to have control of the bus (8:297).

3.2.2 The Domain of Interest.

The domain of logic circuits is quite different from the domain of avionics software for which Tracz' process was developed. Tracz was interested in identifying the components within a given set of applications, whereas the logic circuits primitives are fundamental components that can be composed into an unlimited number of different applications and higher-level components. A second difference is that the objects in the domain had already been defined for the most part. The objective was to modify those components to execute in a different mode.

The existing logic circuits domain defined by Randour and Anderson consisted of 10 logic devices and 2 input/output (I/O) devices. The logic devices represent SSI and MSI components from a single family: they can be composed directly without interface support; thus, their input and output values can be expressed as simple boolean values. None of the devices have tri-state outputs. A brief description of the existing devices is given below, and a more complete discussion can be found in (1).

1. 2-input AND gate: The gate's output is the logical AND of its two inputs.
2. 2-input NAND gate: The gate's output is the logical NAND of its two inputs.
3. 2-input OR gate: The gate's output is the logical OR of its two inputs.
4. 2-input NOR gate: The gate's output is the logical NOR of its two inputs.
5. NOT gate: The gate's output is the complement of its input.
6. 2-bit COUNTER: The counter is incremented whenever its input is true. When the count exceeds a user-specified maximum value (from one to three), the counter resets to zero. The counter can also be reset to zero by asserting its reset line. Two outputs represent a binary expression of the count.
7. 3-to-8 DECODER: Exactly one of eight outputs, as determined by the values on the three select lines, is set to true. The truth table for this device is given in Table 1.
8. 4-input MULTIPLEXER: The output is the value of one of the four inputs; the particular input is determined by the values on two select lines. The truth table is depicted in Table 2.
9. JK FLIP-FLOP: A clocked device whose output is determined by its present state and the values on the two (J & K) inputs. The truth table is shown in Table 3

Table 1. Truth Table - 3-to-8 Line Decoder

X	Y	Z	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Table 2. Truth Table - 4-Input Multiplexer

s_1	s_0	Output
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

10. HALF-ADDER: The SUM and CARRY outputs reflect the result of a binary addition of two boolean inputs. The truth table is shown in Table 4.
11. SWITCH: When the switch is "ON", the output is true; when the switch is "OFF", the ouput is false.
12. LED: The light emitting diode outputs a message of "ON" or "OFF" to the console when its input is true or false, respectively.

3.2.3 Block Diagram. It is the nature of simple logic circuits that the output of any device can be an input to any other device. Therefore, no special relationships exist between devices. Since all devices have boolean inputs and outputs, the domain is *closed* and does not require external inputs and outputs. Because of these factors, a block diagram of this domain was not required.

3.2.4 User Needs. The primary users of a technology base such as this are application specialists designing circuits with domain-oriented application composition systems similar to Architect. The application specialist needs a set of objects whose behavior is predictable and consistent with their real world hardware counterparts. The objects must be modeled with sufficient detail to capture the behaviors that are important to the application specialist. The objects' parameters should be configurable so that a single

Table 3. Truth Table - JK FLIP-FLOP

clock	J	K	New Q	New $\bar{Q}$
0	x	x	old Q	$\sim (oldQ)$
1	0	0	old Q	$\sim (oldQ)$
1	0	1	0	1
1	1	0	1	0
1	1	1	$\sim (oldQ)$	old Q

Table 4. Truth Table - Half Adder

X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

object can be instantiated with a range of behaviors. Objects in the domain should have compatible inputs and outputs so they can be readily composed into applications. When the applications are composed in a graphical environment, iconic representations of the objects are also needed. Icons should be consistent with conventional symbologies used to represent the objects. Cossentine discusses visualization requirements for domain-oriented application composition systems in (4).

As Architect is in the proof of concept phase, a secondary set of users of this technology base are the other researchers who are developing the application executive and modifying the architectural model (25) (6). Both require objects to test and validate their work. The event-driven logic circuits domain, therefore, needs a variety of objects with a range of behaviors sufficient to exercise the event-driven functionality being developed for Architect. An initial assumption was that the 12 devices in the existing technology base would be sufficient for this research.

3.3 Stage 2: Define/refine domain-specific concepts/requirements

The main product of Tracz' second stage is an object-oriented analysis of the domain "with special emphasis on 'identifying commonalities' and 'isolating differences' between applications in the domain" (23:4). The goal in this stage was to identify the domain objects and their attributes, functions, control flows, data flows, and relationships.

3.3.1 Object Model. Normally, the first step would be to develop an object model showing the domain objects and relationships. For the circuits domain, such an object model would be uninformative since the domain objects are already known and the objects have no relationships until composed into an application. Instead, an object model showing the relationships between an arbitrary (domain-independent) event-driven domain and the Architect/OCU architecture was developed. This model, developed in close coordination with Welgan (25) and Gool (6), helped determine the concepts and requirements needed to implement the new domain. The techniques of Rumbaugh, as explained in (21), were used to create the model.

Figure 4 shows that an application consists of one or more composition units and an executive. Composition units (an abstract class) connect to other composition units and can either be objects or subsystems; subsystems are in turn composed of one or more objects. Subsystems also contain an import area and an export area. These parts of the diagram describe the general OCU model. In order to execute in the event-driven mode, the executive needs to maintain a list of events. It also needs a clock to keep track of the passage of time so the time-dependent primitives can execute at the proper time. Welgan designed the executive as a specialized OCU subsystem that both manages and responds to events. Seven different event types were identified to control the execution of an application. The Start and Stop events control the beginning and ending times of execution. The Transmit and Receive events are used to control the exchange of data between top-level subsystems (25). These four event types are *executive* events. The remaining three events, Update, Set-State, and Remove, are *application* events. The distinction between application and executive events is somewhat artificial since the executive, as a subsystem with objects, is controlled by events just as the domain application subsystems are. Events are sent to and collected from the In-Event and Out-Event areas of the top-level subsystems by the executive.

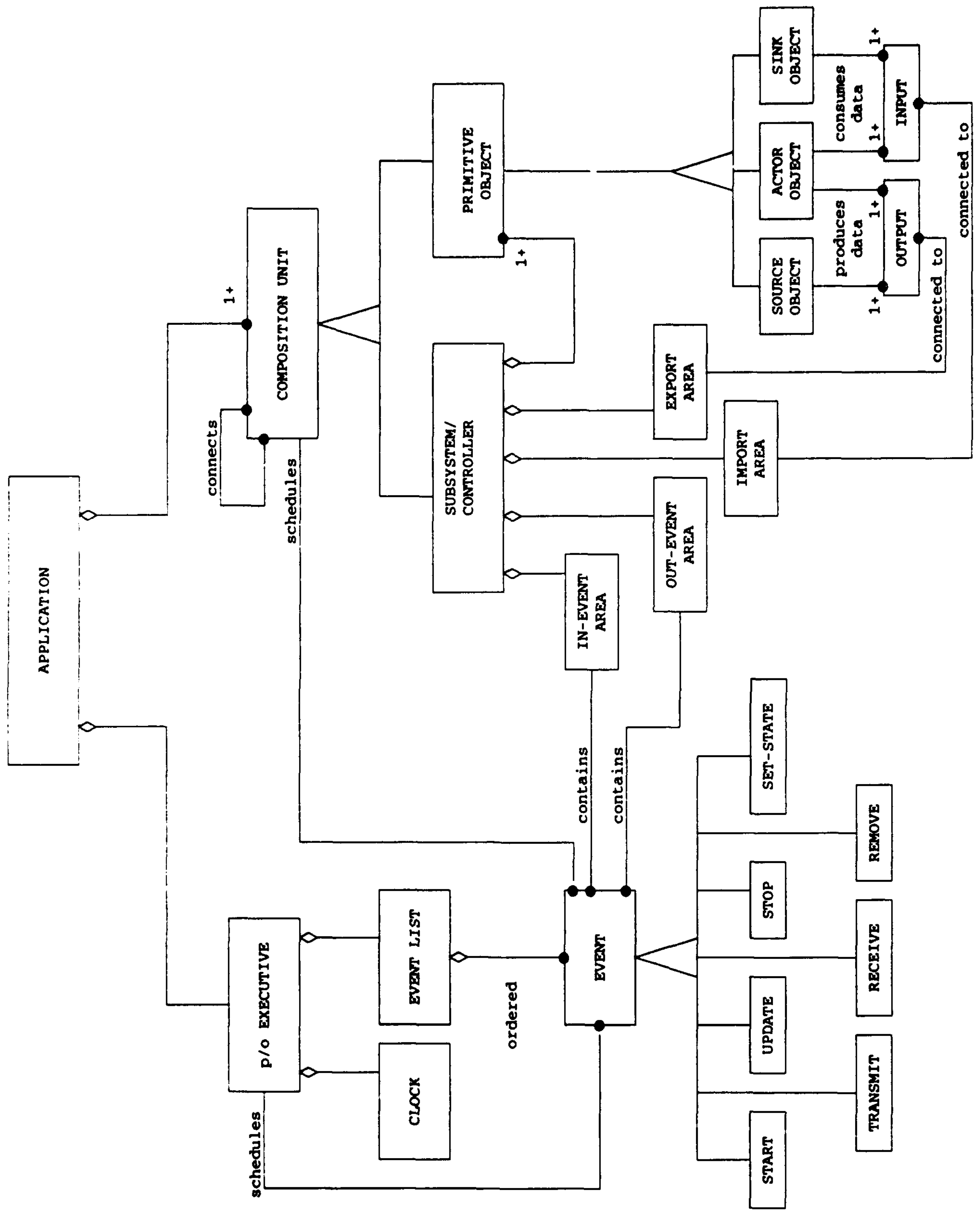


Figure 4. Block Diagram of Event-Driven Domain in Architect

3.3.2 Concept of Operation. The object model defined the relationships between the OCU/Architect architecture and the domain objects and subsystems. The next step was to define a concept of operation that met the following general requirements.

- *Must be compatible with the OCU model.* The OCU model provides only a sparse description of the Update and SetState functions. In describing an object's functions, Lee tersely states that the Update function "calculates new object state data" and the SetState function "(modifies) the objects[sic] state data directly (11:20)." Despite this vagueness, the concept of operation should hold to the spirit of the OCU model to the extent possible.
- *Must maintain temporal consistency of data.* Objects should not update their export data until the time the data is valid; consumers should never have access to data that is not yet valid. This requires objects to maintain consistency between their internal state and the external expression of that state. Failure to meet this requirement will cause erroneous execution or require the executive to be able to detect and handle rollback conditions.
- *Should not be overly complex.* The existing implementation of Architect is elegant in its simplicity. Major architectural changes should be avoided if possible.

Figure 5 shows a simplified functional model of an arbitrary domain object and its interfaces to the architecture/executive. In accordance with the OCU model, the domain object, contained in the dotted lines, has Update and SetState functions and a set of Attributes. When an object is updated, it invokes the architecture function *get-import* to get its new input data. Using this input data and its internal state data and coefficients, the object calculates a new state. In the existing Architect implementation, the next step would be to invoke the *set-exports* function to directly update the object's export data. With the incorporation of delays, a different approach is required.

The Update function determines the delay associated with the new state and invokes the *generate set-state events* to create a set-state event for itself at the proper relative time. Objects do not know about absolute time, only offsets from the current time. The set-state event is then passed to the executive for insertion into the event list. After the delay time

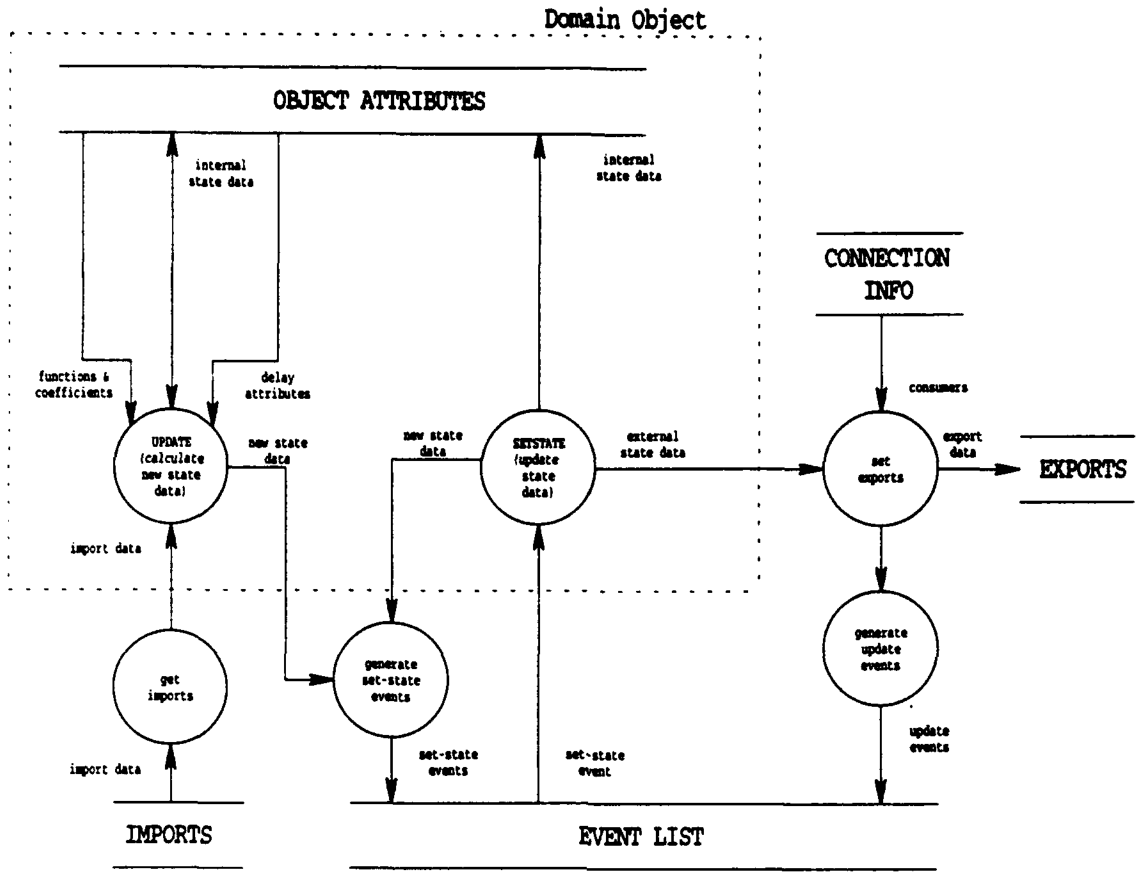


Figure 5. Simplified Functional Model

has expired and the set-state event is processed by the executive, the set-state event is sent to the object's SetState function. The SetState function updates the object's internal state and invokes the set-exports function to update its external state. The SetState function can also cause new set-state events to be created if the particular object requires it.

The set-exports function is also involved in implementing the stimulus-response paradigm. When the set-exports function updates an export value, it determines which objects consume that export data and invokes the *generate update events* function to create Update events for those consumers. These Update events are sent to the executive for insertion into the event list.

When the executive processes an Update or SetState event, it determines which top-level subsystem the intended object is located in. It then places the event in that top-level subsystem's In-Event area and invokes the subsystem's Update function. When the top-level subsystem is updated, it retrieves the new event from its In-Event area and determines if the event is for one of its own objects. If not, it places the event in the In-Event area of its appropriate subordinate subsystem and invokes subordinate subsystem's Update function. The event is passed down the subsystem tree in this way until it reaches the parent subsystem of the event's intended object.

The parent subsystem then invokes either the object's Update function or its SetState function, depending upon the event type. The object's function may return new events. The returned events are passed back up the subsystem tree to the top-level subsystem and on to the executive where they are inserted into the pending event list.

3.3.3 Stale Events. The Remove event is used by objects to notify the executive to delete some previously scheduled event from its list of pending events. In event-driven simulations, an event scheduled for some future time can become "stale" if a subsequent event occurs that cancels or otherwise invalidates the originally scheduled event. For example, consider a billiards simulation in which two balls, B1 and B2, are headed on collision courses. Assume a "collision event" between the two balls is scheduled for the appropriate future time and location based upon distances, relative velocities, rolling resistance, and any other modeled parameters. Now suppose that before that collision occurs, a third ball, B3, impacts B2 and deflects B2 from its course. The previously scheduled collision is now obsolete because B1 and B2 will no longer collide.

In order to execute properly, an application must be able to detect stale events. Stale events can either be detected by the executive or by the domain primitives. Welgan's executive requires the application to tell it when an event needs to be removed (25). A more sophisticated executive might be able to figure out when events have become obsolete and delete them without assistance from the application. This ability would require domain knowledge that a general purpose (domain-independent) executive would not have.

One way to ensure that invalid events are not processed is to give the primitive the knowledge required to know whether or not to process an event that it receives. When the object detects that an event is stale, it simply ignores it. The executive does not become involved. A second approach is to generate a Remove-Event whenever the Update function detects a condition that makes a previously scheduled event obsolete.

3.3.4 Dynamic Models. The next step in Rumbaugh's OOA process is to determine the state models of the domain objects. Figure 6 shows the general dynamic model for the gates, multiplexer, decoder, and half-adder. These objects have no internal state variables; their outputs depend only on the inputs. These objects have a common dynamic behavior and so were combined into a single diagram.

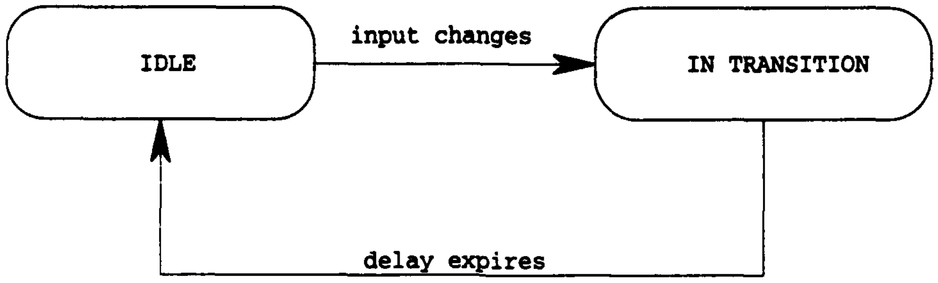


Figure 6. General Dynamic Model for Objects Without Internal State

The *idle* state is a steady-state condition where a device's inputs and ouputs are consistent. When one or more input signals change value, the inputs and outputs become temporarily inconsistent due to propagation delay. During the delay period, the device is *in transition*. Upon expiration of the delay, the outputs change to reflect the new inputs and the device again becomes idle. Devices with multiple outputs can have unique delay times for each output. If more than one output changes as a result of a change of inputs, then the device would remain in transition until the end of the longest delay time of the outputs that change. Under the concept of operation, a device is *in transition* if there is a pending endogenous set-state event for the device. When all pending set-state events have been processed, the device returns to idle.

Figure 7 and Figure 8 show the state diagrams for the counter and switch, two devices having internal state variables. The switch has one binary state variable, position,

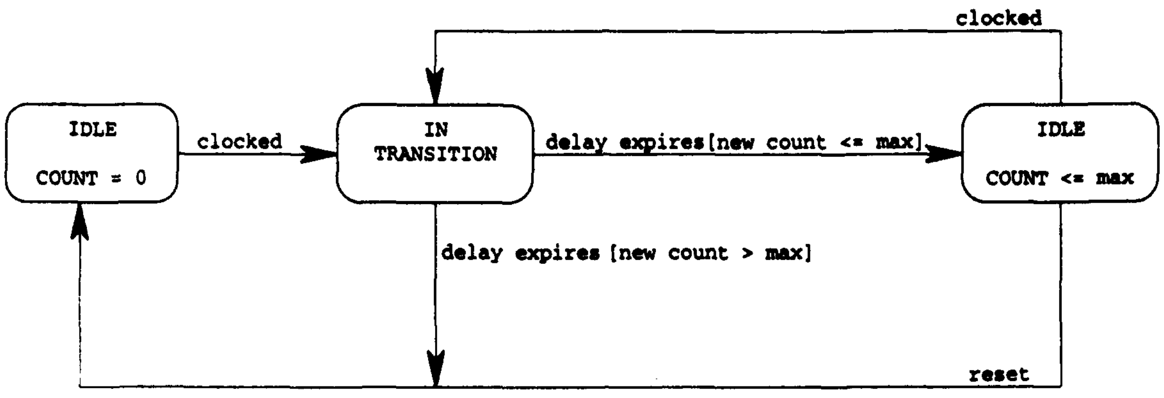

clocked
IDLE
COUNT = 0
clocked
IN TRANSITION
delay expires[new count <= max]
IDLE
COUNT <= max
delay expires [new count > max]
reset

Figure 7. Dynamic Model for the Counter Object

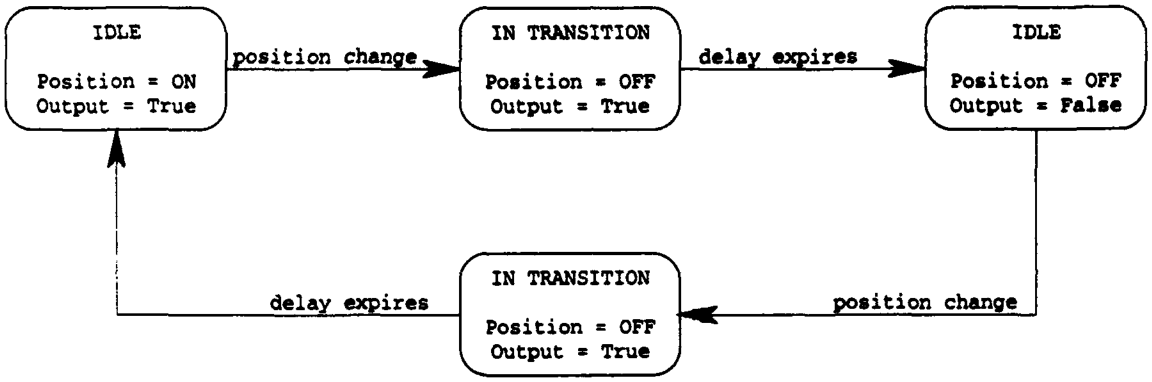
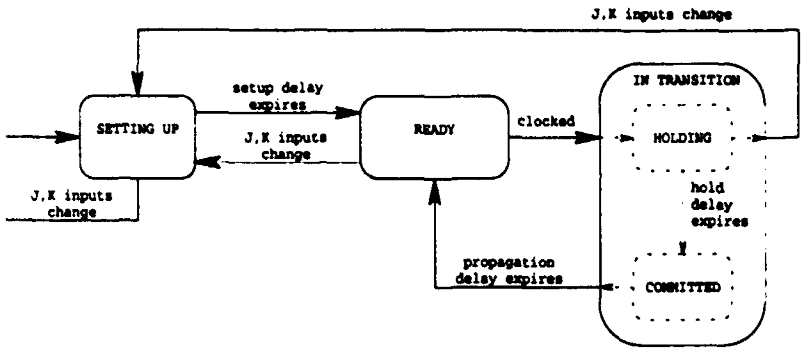


Figure 8. Dynamic Model for the Switch Object

which can be on or off. The existence of a pending set-state event serves as a second pseudo-variable which gives the switch its four states.

Figure 9 shows the state and transition diagrams for the JK-Flip-Flop. Note that the transition diagram differs from that shown for the sequentially executing flip-flop in Figure 3 because this event-driven flip-flop is positive edge-triggered. In addition to propagation delay, the jk-flip-flop also has setup and hold delays. Set-up and hold delays are examples of the VHDL inertial delay. Set-up delay defines the length of time that inputs must remain stable *before* the clock pulse arrives. Hold delay defines the length of time the inputs must remain stable *after* the clock pulse arrives. The jk-flip-flop has three states: *ready*, *setting-up*, and *in-transition*.



State Transition Table

CLOCK	J	K	New Q	New $\bar{Q}$
0	x	x	old Q	~(old Q)
$_0\uparrow^1$	0	0	old Q	~(old Q)
$_0\uparrow^1$	0	1	0	1
$_0\uparrow^1$	1	0	1	0
$_0\uparrow^1$	1	1	~(old Q)	old Q

Figure 9. Dynamic Model for the JK-Flip-Flop Object

Ready is a stable state where the device is in equilibrium, i.e., there are no endogenous set-state events pending. When ready, a change in the J or K inputs moves the the flip-flop to the setting-up state. When the setup delay expires, the flip-flop re-enters the ready state. Alternately, from the ready state, a false-to-true transition on the clock input causes the flip-flop to enter the in-transition state. In-transition is a superstate consisting of *holding* and *committed* substates. Recall that in-transition indicates there is a pending set-state event to change the device's output. If no changes occur on the J or K inputs before the hold delay expires, the set-state event is honored and the output changes after the propagation delay time. If a change occurs on a J or K input prior to the hold delay expiration, the set-state event to change output is not honored and the flip-flop goes to the setting-up state. This requires the device to be able to detect and handle obsolete events.

Analysis of these state diagrams revealed that all the objects have *bistable* outputs; whether high or low, their outputs do not change in the absence of external stimuli. There is a latency from the time the stimulus arrives until the output changes, but once the output changes, it is stable. Other devices have different properties. A monostable multivibrator,

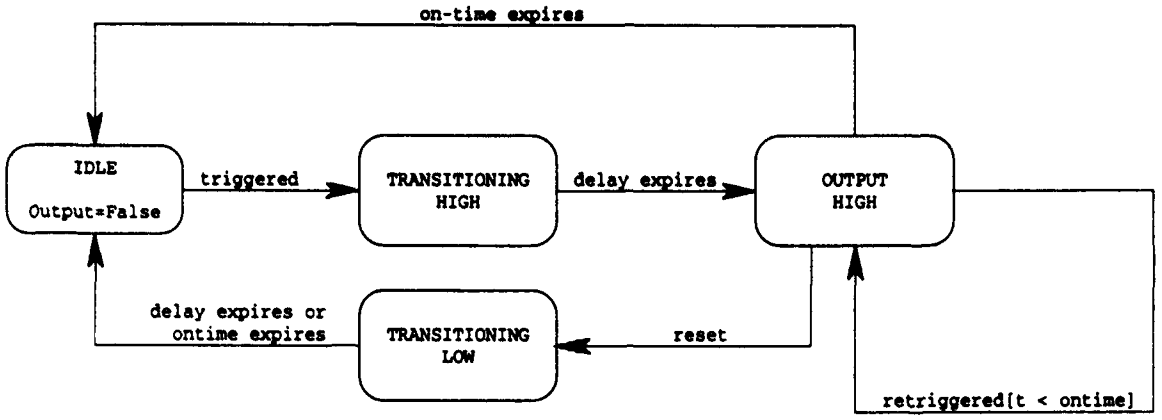


Figure 10. Dynamic Model for the One-Shot Object

or one-shot, is only stable when its output is low. When its outpu: is set high, its new state is transient; it will automatically revert low after a given delay. An astable multivibrator, or clock, has no stable output; it continually alternates between low and high at a periodic rate. In order to examine how transient and periodic behaviors affect design, a one-shot and a clock were added to the domain. The clock will also provide the means to create continuously running applications. In the absence of feedback connections, applications built with the other primitives terminate after a finite number of events have occurred and the primitives are left in a final state. Also, the clock is a fundamental building block that will be needed when more complex LSI and VLSI type components are developed for Architect. The dynamic models of the two new devices are shown in Figure 10 and Figure 11. Rescoping the domain is an example of the iterative nature of Tracz' process; as new requirements are identified, any or all stages may need to be revisited.

3.3.5 Functional Models. The next step in the OOA process is to generate functional models (data flow diagrams) of the domain objects. However, the functionality of the logic circuits devices was too simplistic to warrant development of functional models.

3.4 Stage 3: Define/refine domain-specific design and implementation constraints

This stage identifies constraints on the software architecture imposed by the requirements of domain applications. Architect has no performance requirements, since it is not intended to be a simulation or operational system. Correctness of specified behavior is

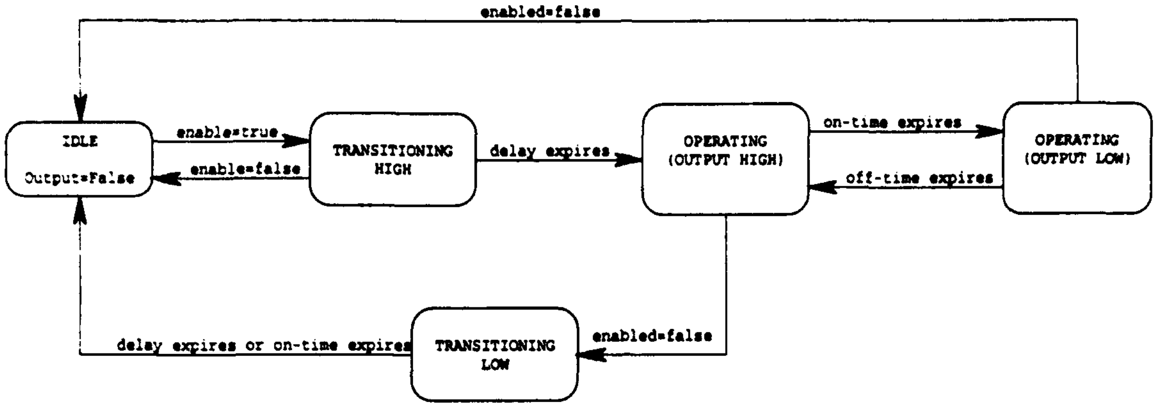


Figure 11. Dynamic Model for the Clock Object

what is important, not how fast or efficiently it executes. Since it is a goal of this research to retain compatibility with the OCU model, this stage of the process should identify domain-independent requirements that cannot be met with the existing Architect/OCU architecture and executive. Many of these requirements were identified previously when the object model and concept of operation were developed. In order to support event-driven, time-dependent domains, the follo' ing changes to Architect are needed.

1. The architecture must be modified to incorporate events.
2. The domain-independent SetState function must be relocated to the object level.
3. The architecture must support the stimulus-response paradigm.
4. The architecture must provide a means to invoke domain-specific semantic checks.
5. An event-driven executive with a clock is required.
6. The architecture must provide for establishing an initial condition prior to execution.

3.5 Stage 4: Develop domain architectures/models

The goal of this stage was to determine the architecture needed to support applications in this domain. The new architectural requirements identified in the previous stage were conveyed to Gool for implementation. Gool extended the OCU model by adding In-Event and Out-Event areas to the subsystems. A diagram of the new OCU subsystem model is shown in Figure 12.

A complete description of the architectural modifications can be found in (6). Because Architect's architecture is domain-independent, this stage also needs to identify

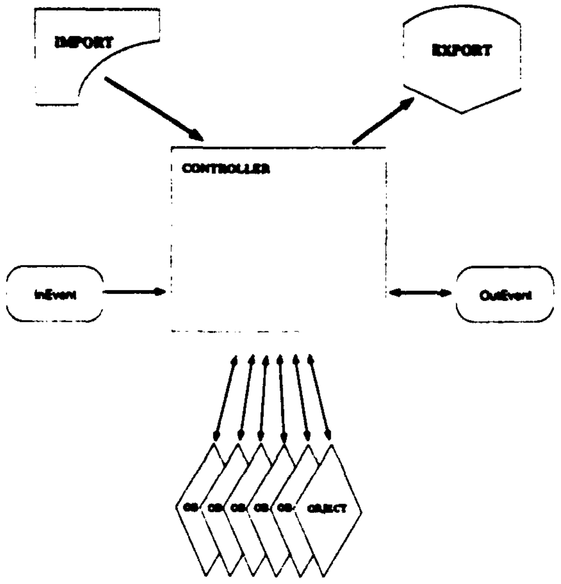


Figure 12. Modified OCU Subsystem with Events

domain knowledge that must be made resident in the domain objects rather than in the DSSA. Sensitivity and domain-specific semantic checks are two types of domain knowledge that normally would be allocated to the DSSA.

3.5.1 Sensitivity. Knowing whether or not to respond to an input signal is device-dependent (and, therefore, domain-dependent) knowledge. In VHDL, objects (entities) only respond to inputs that are listed in their sensitivity list. If a signal arrives on an input listed in the sensitivity list, the appropriate process (function) with the object will be invoked. If the input is not on the list, the input arrival is ignored. VHDL is only a design language, not an implementation of a simulation system; no assumptions can be made about how the language constructs might actually be implemented. Conceptually, however, the decision on whether or not to invoke an object's process is made *outside* the object. Any decision made outside the domain object requires some architectural involvement. If a property is common to all domains, then it is domain-independent, and special architectural support might be justified. Since this was the first event-driven domain developed for Architect, the experience base was not broad enough to determine if sensitivity was a property specific to the domain of digital logic circuits or if it was a domain-independent characteristic. Therefore, it was decided to take the conservative approach and embed the sensitivity knowledge *inside* the domain objects. Under the concept of operation, objects are updated whenever *any* input changes value. Objects with sensitivity then need to keep track of the last state of their sensitive inputs so they can determine when changes occur on those inputs.

3.5.2 Domain-Specific Semantic Checks. This section begins with a discussion of how Architect performs syntax and semantic checks on composed applications. It then discusses how domain-specific semantic checks were incorporated into Architect.

3.5.3 Architectural Syntax and Semantic Checks.

3.5.3.1 Architectural Syntax Checks. Architect enforces the following syntax rules on how components can be connected:

- only input-to-output or output-to-input connections can be made.
- the categories of a connected input and output must be the same.
- the data types of a connected input and output must be the same.

In the logic circuits domain of Randour and Anderson, all inputs and outputs are of category *signal* and data type *boolean*. Thus any output can be connected to any input. This implementation was satisfactory for this part of the research.

3.5.3.2 Architectural Semantic-Checks. Architect also performs semantic checks on the composed application to ensure that the OCU structure is correct. Architect's syntactical and semantic checks allow broad freedom to compose components in many ways. A domain, however, may have its own set of semantic rules that limit the legal compositions to a subset of those allowed by Architect. For example, Architect allows any output to be connected to as many different inputs as the application specialist selects. This may not be realistic in some domains. The output of a logic circuit can only drive a limited number of inputs and still maintain its logic threshold level. The actual number, the fan-out, is device-dependent (and therefore, domain-dependent). Another domain may require certain components be used together; for instance, in domain X, all legal applications containing component A must also contain component B. Prior to this research, Architect had no provisions for domain-specific semantic checks.

3.5.4 Implementation of Domain-Specific Semantic Checks. A software architecture composition tool needs a standard interface to the technology base to determine if domain-specific semantic checks exist. If semantic checks exist, the composition tool then needs to know how to invoke them. The following approach was taken in this research.

In the domain model for each domain, a variable called *<domain-name>-semantic-checks* is declared where domain-name is the actual name of the domain. If domain-specific semantic checks exist for this domain, the value assigned to the variable is the name of the function that needs to be invoked to perform the checks. If no domain-specific semantic checks have been identified, the value of the variable is left as UNDEFINED. In the circuits domain, this variable is declared as:

```
var circuits-semantic-checks : symbol = 'CIRCUITS-SEMANTIC-CHECKS
```

After the architectural semantic checks are completed, the domain-specific checks are invoked using the lisp *funcall* function and the global variable **CURRENT-DOMAIN**. The reason for not using a fixed variable name like "domain-semantic-checks" is to accommodate future applications containing objects from multiple domains in which case domain-specific semantic checks from each domain may need to be invoked.

The domain-specific semantic checks for the circuits domain consist of counting the number of inputs connected to the outputs of each device having a defined fan-out limit. If no object has connections exceeding its fan-out, the semantic checks pass; otherwise, it fails and gives an appropriate error message.

3.6 Stage 5: Produce/gather reusable workproducts

In this stage, the design was mapped into Architect. The previously identified domain objects were implemented as reusable components using Architect's native language, Refine. The 12 existing circuits domain objects were modified to execute in the event-driven mode and the new clock and one-shot objects were developed. All of the requirements identified for the event-driven logic circuits technology base were incorporated into the design.

3.6.1 Object Functions. No new functions were identified for the objects. The existing functions in the OCU model were found to be adequate for the design. It was necessary, however, to modify Architect's implementation of the SetState function from domain-independent to object-specific, which is consistent with the OCU model. In the sequential execution mode, the SetState function is not coupled to the set-export function. When a primitive's state attribute is changed with the SetState function, the internal representation of its state changes, but the external representation does not change until the next time the primitive's Update function is called. Since the sequential execution mode requires a primitive to be updated before any other primitives depending upon its output are updated, no inconsistencies occur. In the event-driven mode, Update functions do not occur as conveniently, and it is necessary to enforce internal and external consistency.

Each primitive was given its own unique SetState function. In the logic circuits domain, four types of domain knowledge are contained within the SetState function.

- Knowledge of the external representation of internal state.

  This knowledge is needed to ensure internal and external consistency whenever the state of an object is modified directly. For example, setting the 2-bit counter's internal count to three is equivalent to setting its two outputs true. Whenever the count is changed, the outputs must be changed to reflect the new state. A domain-independent SetState function would not have the knowledge to perform this action.

- Knowledge of the stimulus-response paradigm.

  Objects in this domain require their Update functions to be invoked only when one or more inputs change value. The SetState function does not "set" outputs that do not change as a result of the new state. Thus in the case of the counter above, either none, one, or both of the outputs might be set True depending upon the previous count.

- Knowledge of latent, periodic, or other time-dependent behaviors.

  When the switch's position is changed, there is a delay before the effect is known externally. The SetState function knows about this latency and schedules a transition event for the output at the proper time. The SetState function will also schedule transition events for primitives whose state is periodic. For example, the clock is a free-running device; once enabled, it requires no further updates to continue running. Whenever the SetState function sets the output, it also schedules an event to toggle the output after the appropriate delay. SetState works similarly for the transient behavior of the one-shot.

- Knowledge to recognize stale events

  As discussed previously, scheduled events sometimes become invalid. An event scheduled to set the clock's output to True, for example, can become invalid if the clock is disabled before the event arrives. Whenever this possibility exists, the SetState function checks the current state of the primitive before processing the event.

It should be noted that knowledge of the stimulus-response paradigm is not *required* for proper execution of an application. Objects' update functions are designed so that if an Update function is called when no inputs have changed, the result will be the same as if the Update function had not been called. The effect of the knowledge then is not to influence application behavior, but to reduce the system overhead by not generating unnecessary events.

3.6.2 Object-Class Hierarchies in the Logic Circuits Domain. In a broad sense, most components in the logic circuits domain have a similar function: they "act" upon their inputs and update their outputs. But two components are fundamentally different. The switch is unique in that it has no inputs. The switch is a *SOURCE* object that exists to provide inputs to other components. Similarly, the LED is unique in that it has no outputs. It is a *SINK* object used to monitor the output state of other components. These differences suggested an object class hierarchy in which primitives belong to one of three subclasses: ACTOR, SOURCE, or SINK as depicted in Figure 4. Further decomposition into lower-level subclasses did not seem appropriate due to the limited number of objects in the domain. If the domain is populated with more objects in the future, a more hierarchical structure may prove beneficial. For example, the ACTOR subclass could be further decomposed into gates, counters, flip-flops, multiplexers, etc. The gates subclass could then have subclasses defined by the number of inputs to each gate. The best hierarchy will depend on the actual objects populating the domain and on the types of applications being composed.

3.6.3 Domain-Specific Language. After the object class hierarchies were established and the primitives were coded, it was necessary to develop a grammar so that applications could be composed and parsed into Architect. Dialect, the Refine environment grammar tool, was used to create the DSL for the circuits domain. A grammar for the original circuits domain already existed, so all that was required was to modify it to include the two new objects and the new attributes added to other objects. An executable application in the time-dependent domain requires three grammars:

- Domain-specific grammar
- Architectural (OCU) grammar
- Executive grammar

Limitations within Dialect restrict a grammar to inherit from at most one other grammar (19:5-20). To circumvent the limitation, the executive grammar was combined with the OCU grammar. The DSL then inherits from the combined OCU/executive grammar (25). After compiling the DSL, the next step was to test the domain objects with simple applications. Validation of the new technology base components is discussed below.

3.6.4 Test Methodology. The following methodology was used to validate the new logic circuits technology base:

1. Validated SOURCE and SINK objects
2. Validated remaining objects by composing simple applications containing previously validated objects and the object being tested. This step was repeated until all objects were validated.
3. More complex applications were composed to test object interaction.

3.6.5 Test Results.

3.6.5.1 Validate SOURCE and SINK Objects. The first step was to test the SOURCE and SINK objects, as these are needed to test all the other objects. An application with one subsystem consisting of a single switch connected to an LED was composed. The switch was defined with initial position OFF and delay 5. Two SetState events for the switch were placed into the executive's event-manager list – one to change the switch position to ON at time zero and another to set it back to OFF at time 10. The application was then parsed into Architect and executed. The sequence of application events described below occurred:

1. The first pre-loaded SetState event is processed for the switch at t=0. The switch changes its position to ON and returns a new SetState event for itself to change its output to TRUE at t=t+5.
2. The new SetState event for the switch is processed at t=5. The switch changes its output to TRUE and returns an update-event for the LED.
3. The update-event is processed for the LED at t=5. The LED updates its display value to ON.
4. The second pre-loaded SetState event is processed for the switch at t=10. The switch changes its position to OFF and returns a new SetState event to change its output to FALSE at t=t+5.
5. The new SetState event is processed for the switch at t=15. The switch changes its output to FALSE and returns an update-event for the LED.
6. The update-event for the LED is processed at t=15. The LED changes its display value to OFF.

Additional events pertinent to the executive's function also occurred but are not listed here since they did not affect the actual operation of the switch and LED. The successful completion of this test validated the behavior of the switch and LED objects.

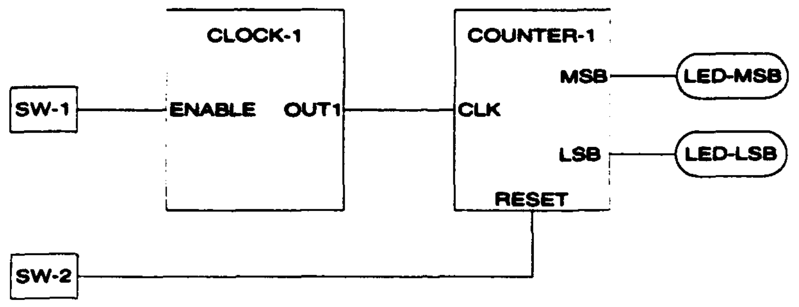


Figure 13. Counter Test

3.6.5.2 Validate Remaining Components. The ability to preload SetState events for the switches into the executive's event-manager provided a simple yet effective way to generate the input transitions needed to test the behavior of the other devices. This was particularly useful when generating the precise sequencing of transitions needed to test the setup and hold delays on the jk-flip-flop. The clock object, after it was validated, provided a simpler way to generate transitions for testing other objects which did not require the precise transitions. The application used to test the counter is depicted in Figure 13.

3.6.5.3 Feedback Circuit. A simple oscillator application was developed to demonstrate the ability to specify applications using feedback. The circuit diagram used for the test is shown in Figure 14. The feedback connection creates an unstable circuit which oscillates with a period determined by the propagation delays through the inverters. Since the circuit does not have any switches and the inverters do not have any state variables, set-state events cannot be used to initiate execution. Instead, an update event is used. When the circuit is composed, there is always one object in the feedback loop where input and output conditions are inconsistent. The update event is targeted for this object.

The test begins with the Not-1 and Not-2 inputs true, and the Not-3 input false. LED-1 and LED-3 are ON and LED-2 is OFF. Not-1 is in an unstable condition because its input and output are the same (both true). The application specialist inserts an Update event for Not-1 into the event manager prior to execution. When Not-1 updates, it sets its output false. This causes Not-2 to update which in turn causes Not-3 to update. Due to

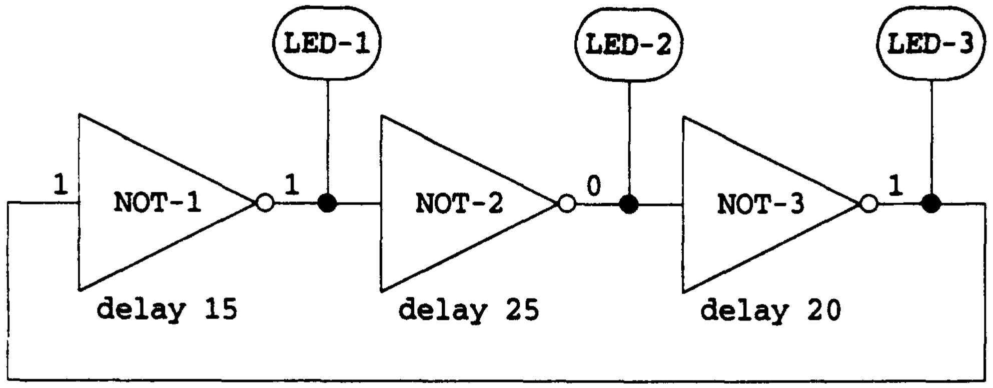


Figure 14. Simple Oscillator

the feedback from Not-3's output to Not-1's input, the circuit will oscillate. The period of oscillation depends upon the cumulative delays in the three gates. With the delays set to 15, 25, and 20 for Not-1, Not-2, and Not-3, respectively, the total delay through the gates is 60. LED-1 will go OFF at t=15, LED-2 will go ON at t=40 (15+20), and LED-3 will go OFF at t=60 (15+25+20). Afterwards, the three LEDs will change state every 60 time units until a STOP event is processed.

Following is the actual output generated when the application was executed:

```
.> (ar 15)
The current simulation time is now  0
The current simulation time is now  15
LED LED-1 = OFF
The current simulation time is now  40
LED LED-2 = ON
The current simulation time is now  60
LED LED-3 = OFF
The current simulation time is now  75
LED LED-1 = ON
The current simulation time is now  100
LED LED-2 = OFF
The current simulation time is now  120
LED LED-3 = ON
```

```
The current simulation time is now  135
LED LED-1 = OFF
The current simulation time is now  160
LED LED-2 = ON
The current simulation time is now  180
LED LED-3 = OFF
The current simulation time is now  195
LED LED-1 = ON
The current simulation time is now  220
LED LED-2 = OFF
The current simulation time is now  240
LED LED-3 = ON
The current simulation time is now  250

Rule successfully applied.
.>
```

3.6.5.4 Comparison of Sequential and Event-Driven Primitives. Table 5 compares the attributes in the sequential and event-driven JK-Flip-Flops, respectively. A 'Yes' means that an attribute was both included and implemented in the primitive; 'No' means that an attribute was included in the primitive but not implemented; a '–' means that the primitive did not contain the attribute. An attribute is considered to be implemented if its value is used by one or more functions within the primitive.

Implementation of time-dependent behavior introduced many more states to the jk-flip-flop, requiring more state variables. The Clock-Level attribute stores the previous values of the clock input in order to model edge-triggered clocking. As the Update function is invoked whenever any input changes, J-Level and K-Level attributes were needed to determine if the changes occurred on the J or K inputs since changes on those inputs affect the validity of the setup and hold delays. Each time the J or K input changes, a 'setup delay expired' event is scheduled. The JK-Changes attribute counts the number of

Table 5. Comparison of JK Flip-Flop Attributes

Attribute	Sequential	Event-Driven
Manufacturer	No	No
Mil-Spec?	No	No
Power Level	No	No
Delay	No	Yes
Setup Delay	No	Yes
Hold Delay	No	Yes
State	Yes	Yes
Fan Out	–	Yes
Clock Level	–	Yes
J Level	–	Yes
K Level	–	Yes
JK Changes	–	Yes
Mode	–	Yes
Hold Delay Expired	–	Yes

times a J or K input changes. After the setup delay for each change expires, the count is decremented. When the count is zero, the inputs are stable and ready to be clocked. (More properly, the Remove-Event should have been used to cancel the previous 'setup delay expired' event; however, the functionality to support Remove-Events was not yet implemented in the executive when the JK-Flip-Flop was developed and tested.) The Mode attribute reflects the dynamic state of the JK-Flip-Flop, i.e., whether it is "setting up", "holding", or "ready" as shown in Figure 10.

3.7 Summary

Fourteen event-driven, time-dependent domain primitives were developed during this research. Twelve of these were modifications to existing non-event driven sequentially executing primitives originally developed by Randour and Anderson (18) (1). The event-driven primitives have greater complexity than the sequential primitives. The primitives now access new architectural functions that allow the primitive to raise events. The new primitives also have an added function – the SetState function – that interprets set-state events and manages the internal and external representation of state.

The domain engineering process adequately addressed most issues needed to implement this technology base. Missing from the process were those issues relating to the domain-specific semantic checks. Because the process is geared toward domain-specific software architectures, it assumes that all domain knowledge not contained within the domain objects will reside in the architecture or executive. This is not a satisfactory solution for the domain-independent architectuie and executive used in Architect. However, the process was not difficult to adapt to the unique Architect environment.

IV. Design and Implementation of the Cruise Missile Technology Base

4.1 Introduction

The objectives of developing the cruise missile technology base were: to create a set of time-driven primitives that are significantly more complex than the event-driven primitives in the logic circuits domain; to learn more about the domain modeling process; to determine architectural impacts/requirements on Architect. This chapter reviews the design and implementation of the cruise missile technology base in the context of Tracz' domain engineering process.

4.2 Stage 1: Define the Scope of the Domain

The main purposes of this stage are to bound the domain of interest and to produce a diagram of the domain showing inputs, outputs, and high-level relationships. User needs are also identified.

4.2.1 Scoping the Domain. Figure 15 shows a high level diagram of the cruise missile domain. In addition to the missile, there are support objects such as the launch platform and maintenance facilities. There are also ancillary functions such as mission planning, targeting, and damage assessment involved in the employment of the missile. In flight, the missile interacts with the atmosphere and is affected by gravity. If the missile has terrain-following capabilities, it must interact with the physical features of the terrain. Some advanced missiles also receive navigational information from satellites. For the purposes of this research, several simplifying assumptions were made as part of defining the domain of interest.

- The domain of interest does not include support objects such as launch platforms, or ancillary functions such as targeting, mission planning, or damage assessment.
- A calm atmosphere was assumed, i.e., wind was not modeled.
- A featureless terrain was assumed, i.e., no obstacles in the missile's flight path.
- No external navigational aids are required.
- Gravity is assumed to be constant.

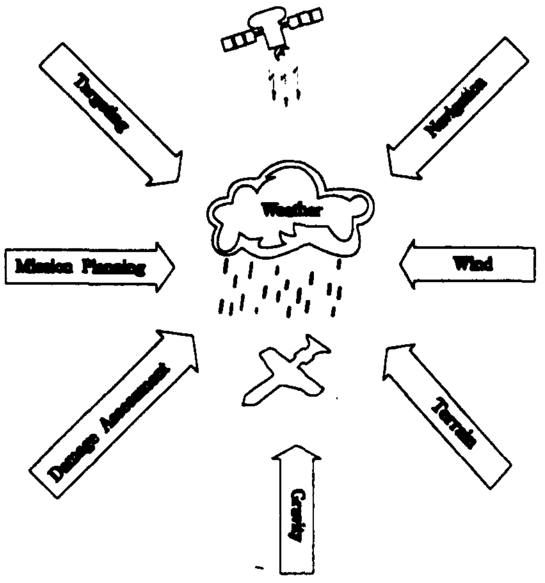


Figure 15. Elements of the Cruise Missile Domain

Based upon these simplifying assumptions, the domain of interest consists only of the missile proper.

4.2.2 User Needs. Many user needs for this domain parallel the needs identified in Section 3.2.4 for the logic circuit domain. Objects should exhibit the behaviors that are of interest to the application specialist. The objects should also be consistent, predictable, and configurable. However, the objects in this domain are more difficult to configure than the logic circuit domain objects. The logic circuit objects are not inter-related and there are few restrictions on how they can be composed into meaningful applications. In contrast, the constituent objects of a cruise missile have relationships that constrain their composition. For example, an engine could be connected to a fuel tank, but it would not make sense to connect an engine to a guidance system. After the application specialist composes the missile, he must assign attribute values to the constituent objects. Relationships among the objects constrain the range of values that can reasonably be assigned. For example, the weight of the missile affects engine power requirements, and the supply of fuel constrains the missile's range. The problem the application specialist faces then, is not how to *compose* the application, but rather how to *configure* the objects once they have been composed. Semantic checks can help identify improper configurations.

Ideally, the application specialist should only have to specify a set of requirements (range, speed, weight, etc.), and the composition tool, using heuristics, would assist the application specialist in assigning appropriate attribute values to meet those requirements. However, such a capability is beyond the scope of this research.

4.3 Stage 2: Define/refine domain-specific requirements/concepts.

In this stage, system requirements and concepts were identified for the domain of interest. An OOA was then performed using the methods of Rumbaugh (21).

4.3.1 System Requirements and Concepts. Two cruise missiles in use today are the Air-Launched Cruise Missile (ALCM) and the Tomahawk sea-launched cruise missile. While their specific capabilities differ, they have several common features. They both have air-breathing engines that consume a liquid fuel. After launch, they are autonomous in flight, requiring no external support from the launch platform or other ground-based facilities, and they fly pre-programmed courses toward a designated target at subsonic speeds. They fly at low altitudes to make detection more difficult (12, 13). These common features were used in the design of the cruise missile model for this research. Even though launch platforms themselves are not part of the domain, there must be a way to program the route and target location and to 'launch' the missile. It is envisioned that these exogenous activities/events will be performed by having the application specialist enter events into an event manager prior to execution.

The purpose of a cruise missile is to deliver ordnance on a target. Starting from an initial location with pre-programmed route and target information, the missile follows the route toward the designated target. The route consists of a sequence of route points. To follow the route, the missile must have a means to determine its current location in reference to the next route point and to alter its course if necessary so that it reaches the desired point. When it reaches a route point, the missile alters its course so that it heads toward the next route point. When the missile turns toward the last route point, it arms its ordnance. Upon reaching the last route point, which is the target point, it detonates its ordnance.

4.3.2 Object-Oriented Analysis. Based on these basic requirements and concepts, an object model of the cruise missile was developed showing the objects, attributes, and relationships. Figure 16 depicts this model. The cruise missile was decomposed into four subsystems: propulsion, avionics, warhead, and airframe. The propulsion subsystem performs the expected function of providing thrust for the missile. The avionics subsystem provides the navigation and guidance functions for the missile. It also provides the steering function to keep the missile's flight path on course. The airframe integrates the thrust and steering commands to determine the missile's actual position, velocity, acceleration, and heading. The warhead subsystem performs the ordnance function.

Each subsystem was then decomposed into a set of objects that collectively perform the function of the subsystem. State diagrams and data flow diagrams were developed for each object in a subsystem. The four subsystems are described below.

4.3.3 Propulsion Subsystem. In order to perform the propulsion function, this subsystem requires a source of fuel, an engine, and a control to regulate the engine thrust. To provide these functions, the propulsion system was decomposed into three components: a fuel-tank, a jet-engine, and a throttle.

4.3.3.1 Fuel Tank. The main function of the fuel tank is to provide fuel to the throttle. The tank was modeled as having an integral fuel pump which must be started before fuel can be delivered to the throttle. As fuel is consumed, the weight of the fuel tank decreases, altering the weight of the missile. The fuel tank's inputs, outputs, and attributes are shown in Table 6.

Table 6. Fuel-Tank

Inputs	Outputs	Attributes
fuel-pump-start	fuel-available?	fuel-capacity
consumption-rate	tank-weight	empty-weight
current-time		fuel-level
		fuel-density
		pump-on?
		last-time
		flow-rate

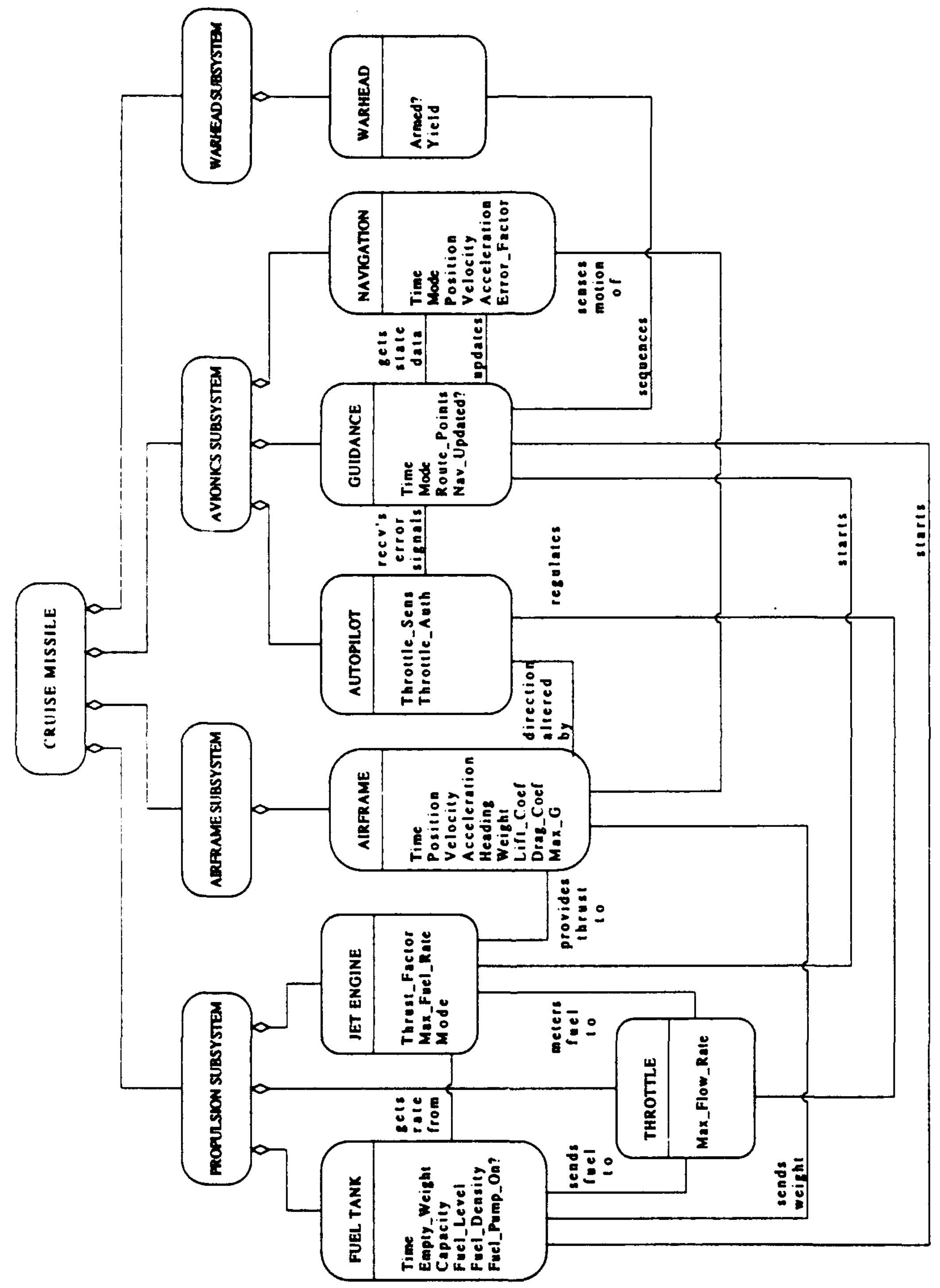


Figure 16. Object Model of the Cruise Missile Domain of Interest

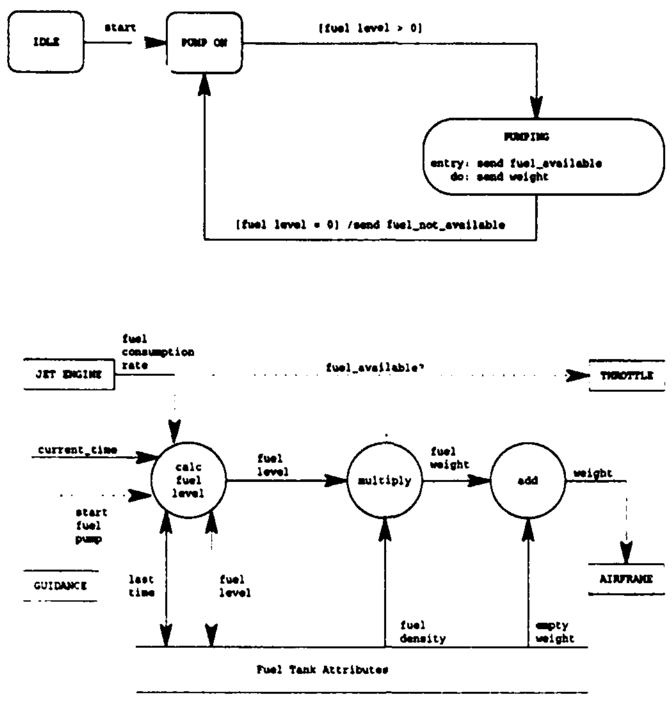


Figure 17. Dynamic and Functional Flow Diagrams for the Fuel-Tank

The dynamic and functional models of the fuel-tank are shown in Figure 17. Fuel-tank is activated when it receives a fuel-pump-start signal from guidance. If the fuel-level is not empty, the fuel-tank outputs a fuel-available signal (simulates fuel pressure) to the throttle. The flow-rate and last-time attributes, in conjunction with the current-time inputs, are used to determine how much fuel was consumed since the last update. From this, a new tank-weight is calculated using the empty-weight, fuel-density, and fuel-level attributes.

4.3.3.2 Throttle. The throttle's function is to meter fuel to the jet-engine. Table 7 shows the inputs, outputs, and attributes required to perform the throttle function.

Table 7. Throttle

Inputs	Outputs	Attributes
fuel-available? throttle-index	requested-flow-rate	max-flow-rate

The throttle's functional model is shown in Figure 18. Throttle receives a throttle-index (a percentage value) from the autopilot. If the fuel-available? input signal from the

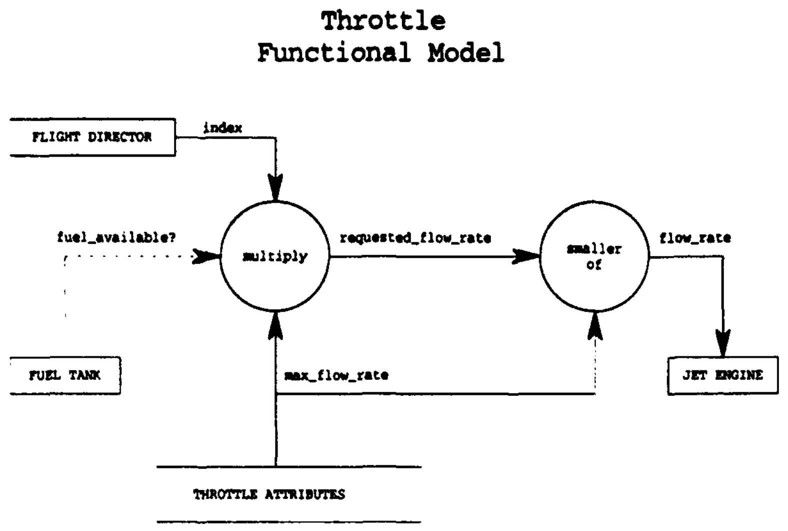


Figure 18. Functional Flow Diagram for the Throttle

fuel-tank is true, then throttle outputs a requested-flow-rate to the jet-engine that is the product of the throttle-index and its max-flow-rate attribute.

4.3.3.3 Jet Engine. The main function of the Jet-Engine is to provide thrust to the airframe. To perform its function, it requires the inputs, outputs, and internal attributes shown in Table 8.

Table 8. Jet-Engine

Inputs	Outputs	Attributes
start inflow-rate current-time	thrust consumption-rate	max-fuel-flow-rate thrust-factor mode

The dynamic and functional models of the Jet-Engine are shown in Figure 19. Jet-Engine is activated when it receives a start signal from guidance. Upon receipt, it sets its mode from "off" to "starting" and waits until the fuel inflow-rate is greater than zero. When this occurs, Jet-Engine transitions its mode to "running" and outputs thrust to the airframe. To help keep the model simple, the output thrust was assumed to be directly proportional to the fuel consumption rate, with the thrust-factor attribute being the constant of proportionality. Thus, thrust is determined by multiplying the consumption-rate (the lesser of inflow-rate or max-fuel-flow-rate) by the thrust-factor. The consumption-rate

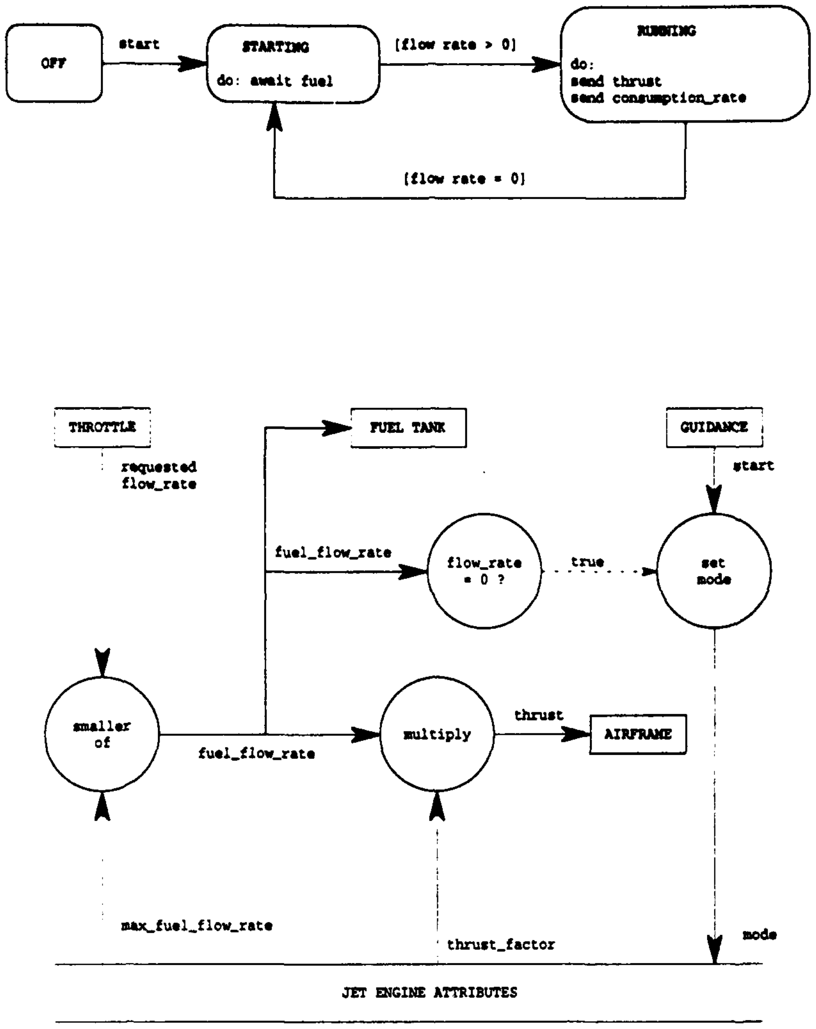


Figure 19. Dynamic and Functional Flow Diagrams for the Jet-Engine

is also output to the fuel-tank so that the fuel-level in the tank can be updated as fuel is consumed by the engine. If the inflow-rate drops to zero while the jet-engine is running, the mode transitions back to "starting", simulating an 'out of fuel' condition.

4.3.4 Airframe Subsystem. The Airframe subsystem consists only of an airframe object. The main function of the airframe is to transform the thrust, heading, pitch, and weight data into new acceleration, velocity, and position vectors. Airframe's parameters are shown in Table 9.

The Airframe's functional model is given in Figure 20. Using its last known acceleration, position, and velocity, the airframe computes a new position and airspeed for

Table 9. Airframe

Inputs	Outputs	Attributes
current-time	position	position
thrust	velocity	velocity
tank-weight	acceleration	acceleration
new-heading	current-heading	heading
new-altitude		turn-rate
		max-g-turn
		lift-coefficient
		drag-coefficient
		weight
		last-time

the current time. It next computes a new heading based upon the current heading, the requested new-heading and its allowable turn-rate. The heading, elevation-rate, and airspeed are then used to calculate new velocity vectors. Next, it computes new acceleration vectors using thrust, weight, lift and drag coefficients, airspeed, and heading. A simplifying assumption was made that drag is proportional to the square of the airspeed.

4.3.5 Avionics Subsystem. In order to perform the avionics function, this subsystem needs to be able to estimate the missile's current state vectors (position, velocity, and heading) and determine the errors from the intended course (2). It must then issue the needed correction commands to the airframe and propulsion subsystems to alter course and speed. To perform these functions, the avionics subsystem was decomposed into Navigation, Guidance, and AutoPilot objects.

4.3.5.1 Navigation. Inertial navigation is based upon measuring the acceleration of a vehicle and then integrating that acceleration to determine velocity and position. The acceleration is integrated twice. The first integration yields velocity and the second integration yields position. The resulting velocity and position are relative to the vehicle's initial velocity and initial position; thus inertial navigation requires that these initial conditions be known (22:4). Any errors in the measured acceleration are also integrated. If the measured acceleration is off by a constant amount, the velocity error will increase linearly with time and the position error will grow at a parabolic rate. Over time,

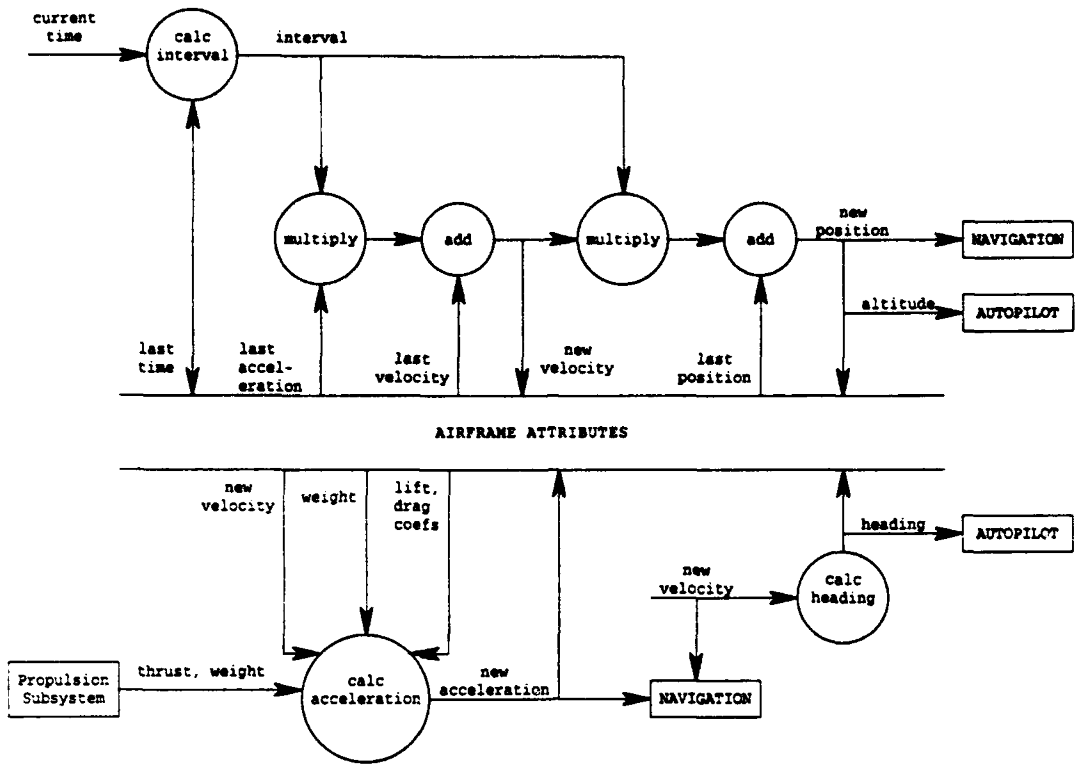


Figure 20. Functional Flow Diagram for the Airframe

even minute acceleration errors can result in unacceptably large (unbounded) position errors. To overcome this limitation, inertial systems often use navigation aids (nav-aids) to improve their accuracy. Nav-aids fall into two categories: velocity aiding and position aiding. Doppler radar is an example of the former; Tactical Air Navigation (TACAN) and Long Range Navigation (LORAN) are examples of the latter. The Global Positioning System (GPS) can provide both (13:179).

The function of the Navigation object is to provide estimates of position, velocity, and heading to the Guidance object. It contains the inputs, outputs, and internal attributes as shown in Table 10.

The Navigation object simulates an inertial system with a type of internal position aiding called terrain contour matching (TERCOM). TERCOM requires that the flight path has been previously mapped and the terrain elevation data stored in a database. During flight, an onboard radar altimeter detects the contours of the terrain beneath the missile, and an onboard processor compares the altimeter returns to the contour information in

Table 10. Navigation

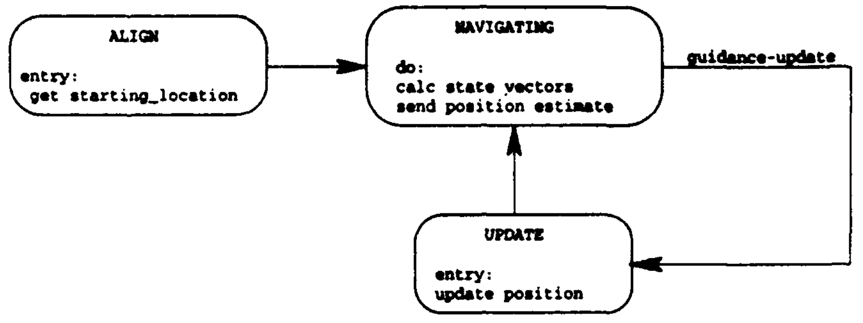
Inputs	Outputs	Attributes
current-time	nav-position	position
missile-acceleration	nav-velocity	velocity
missile-velocity	nav-heading	acceleration
missile-position		error-factor
guidance-update		mode
		last-time

the database. When a match is found, the INS position is updated with the correct coordinates. When the radar altimeter is transmitting, the missile is more vulnerable to detection. Therefore, the number of TERCOM updates is held to a minimum consistent with navigation accuracy requirements (13:220-225).

The TERCOM system is not modeled here, but its effect is modeled by having very low (zero) error in the z-axis estimate of position (i.e., nav estimate of altitude = missile altitude). TERCOM is modeled by navigation receiving position updates (nav estimate of position = missile position) whenever route points are reached. The error-factor attribute is used to generate a random error which is added to the input acceleration, simulating errors in the acceleration measurement. Over time between updates, the errors will cause the INS estimates to drift.

The dynamic and functional models of the Navigation object are shown in Figure 21. The Navigation object has three modes: align, navigate, and update. In the align mode, it sets its internal position and velocity attributes to the values output by the Airframe object. This ensures that it begins with the correct initial conditions. In the navigate mode, it provides position, velocity, and heading estimates which are subject to errors as explained above. In the update mode, it updates its internal position to that of the Airframe object.

4.3.5.2 Guidance. The purpose of the Guidance object is to generate error signals which can be used to alter the heading, altitude, and speed of the missile so that the missile arrives at the proper locations at the proper times. It also controls the sequencing



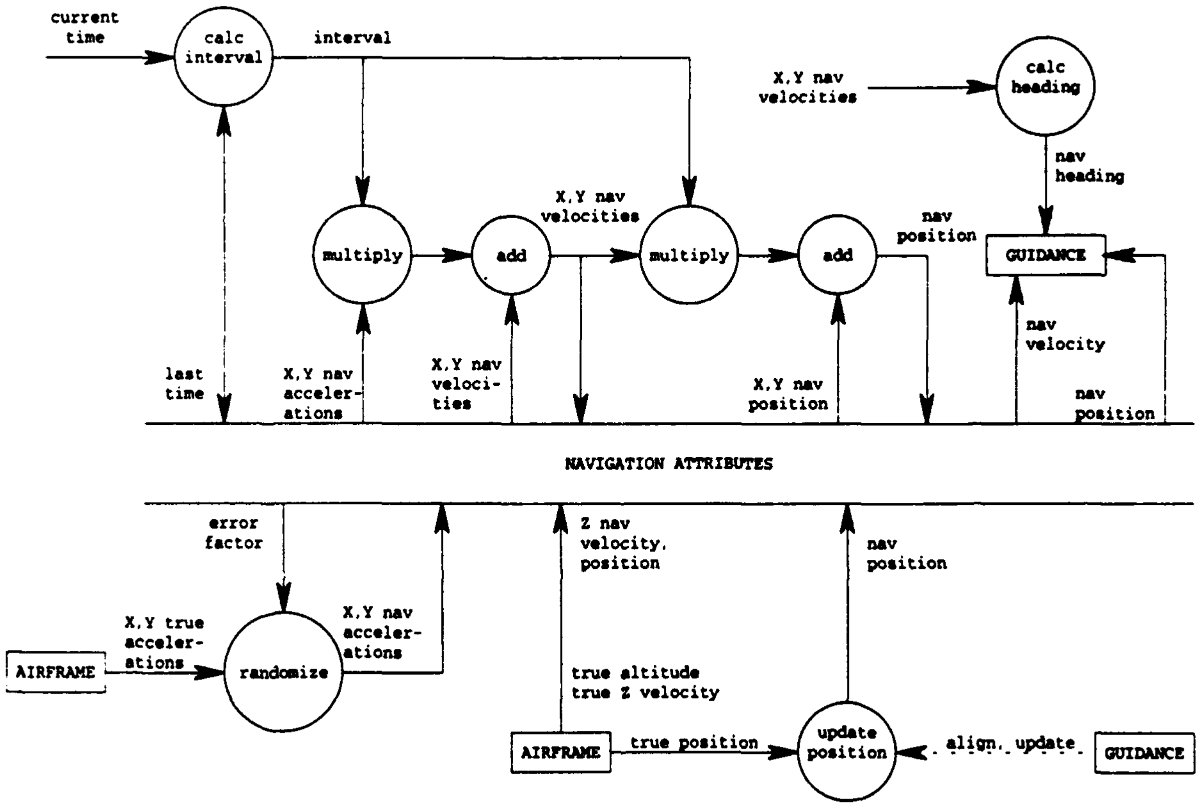


Figure 21. Dynamic and Functional Flow Diagrams for Navigation

of other objects. To perform this mission, Guidance has the inputs, outputs, and internal attributes show in Table 11.

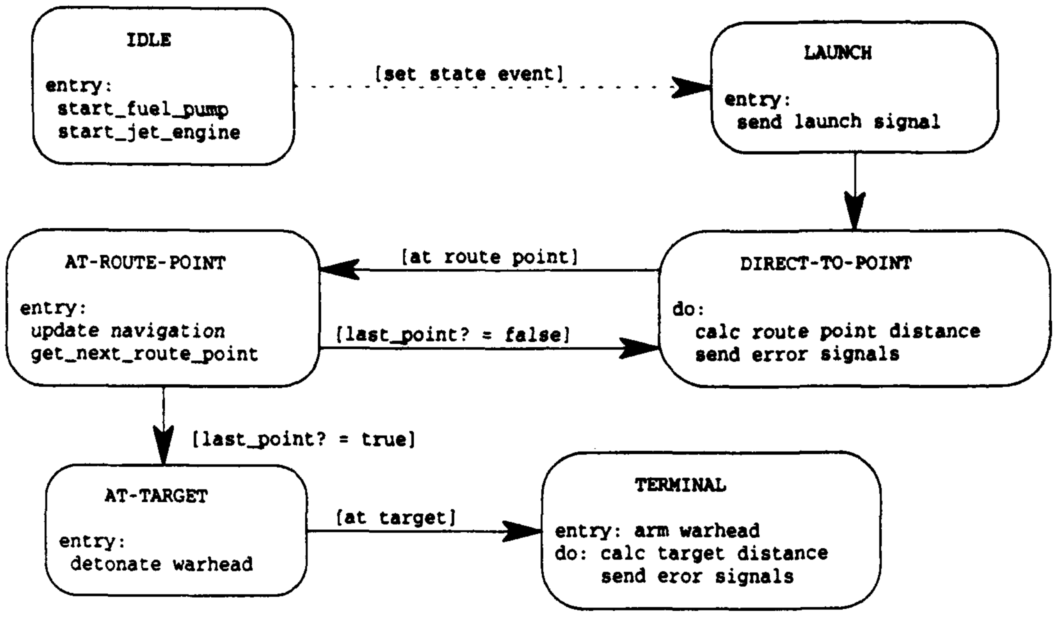
Table 11. Guidance

Inputs	Outputs	Attributes
nav-position	heading-error	route-points
nav-velocity	altitude-error	mode
nav-heading	speed-error	nav-updated?
current-time	start-fuel-pump	
	start-jet-engine	
	arm-warhead	
	detonate-warhead	

The dynamic and functional models of the throttle are shown in Figure 22. The Guidance object has four modes: idle, launch, direct-to-point, and terminal. In the idle mode, it sends a start-fuel-pump signal to the Fuel-Tank object. It then waits for its mode to be changed to launch via a SetState command (the application specialist inserts a SetState event in the event manager prior to execution). When its mode changes to launch, Guidance sends a start-jet-engine command to the Jet-Engine object and transitions to the direct-to-point navigation mode.

Guidance contains a list of route-points that the missile is to follow. Upon entering the direct-to-point mode, Guidance calculates the distance from the current position, as estimated by the Navigation object, with the position of the first route-point in its list. It then calculates the required heading and speed to arrive at that point at the indicated time. Next, it compares these calculated values with current heading, altitude, and speed to determine its heading-error, altitude-error, and speed-error outputs.

In the time-driven mode, the Update function is called on a periodic schedule. However, the actual arrival time at a route-point is asynchronous with the update schedule, i.e., the missile may arrive at the route point between Update calls. Guidance must therefore determine the best time to perform an activity associated with arrival at a route-point: either just before it gets there, or just after it passes. To make this decision, Guidance calculates a projected distance for the next time interval based upon current speed and heading. At the time projected distance exceeds current distance, the missile is as close as



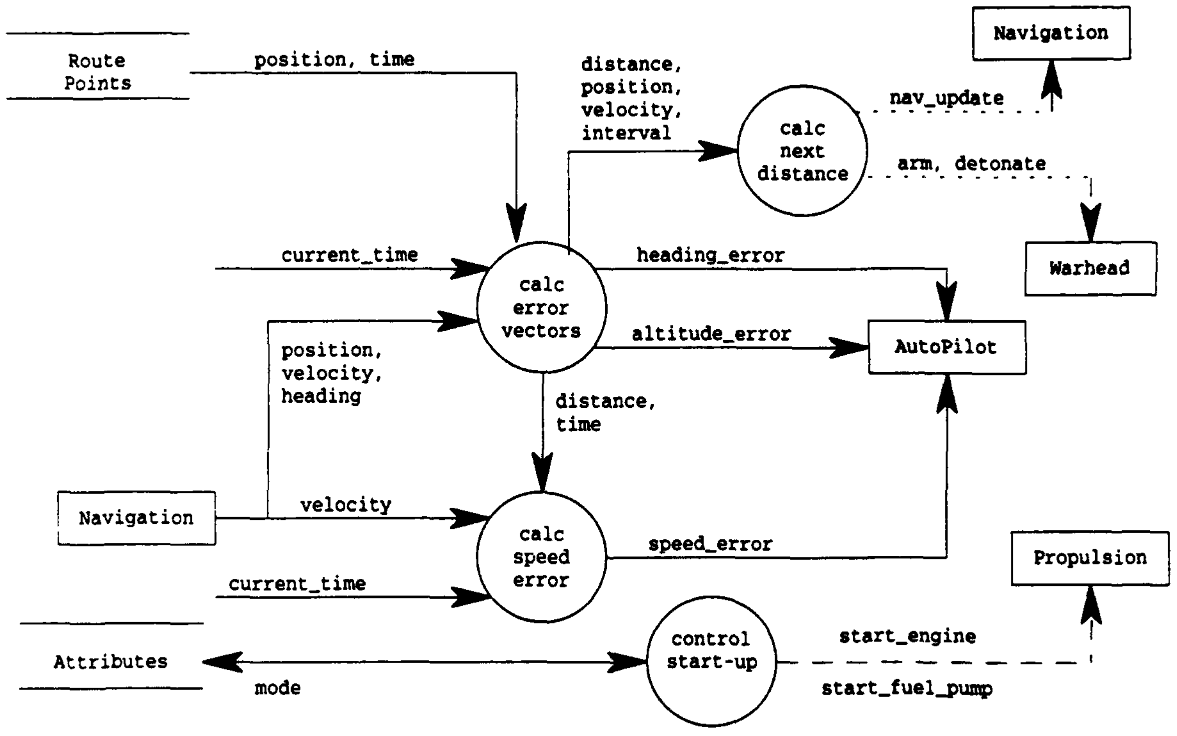


Figure 22. Dynamic and Functional Flow Diagrams for Guidance

it will get and activities associated with arrival are performed at this time. In an execution mode that permits asynchronous events to be raised by the objects, this step would not be required. However, mixed-mode execution is not currently supported in Architect.

When Guidance determines that it has arrived at a route point, it sends a nav-update signal to Navigation, selects the next route-point from the list, and calculates error signals for that new route-point. When the last route-point is selected, Guidance sends an arm-warhead signal to the Warhead object and enters the terminal mode. In terminal mode, Guidance generates error signals as before. When the last route-point (target) is reached, Guidance sends a detonate-warhead signal to the Warhead object.

4.3.6 AutoPilot. The AutoPilot object transforms heading-error and altitude-error signals from the Guidance object into heading and elevation control commands for the Airframe object. It also transforms speed-error signals into throttle commands for the Propulsion subsystem. AutoPilot consists of the inputs, outputs, and internal attributes shown in Table 12.

Table 12. AutoPilot

Inputs	Outputs	Attributes
speed-error	throttle-index	throttle-authority
heading-error	heading-command	throttle-sensitivity
altitude-error	altitude-command	
missile-velocity		
missile-heading		
missile-altitude		
current-time		

Figure 23 shows the functional model of the Autopilot object. The Autopilot object has no interesting states so no dynamic model was developed. Considerable liberty was taken in the design of the AutoPilot object to simplify the way the Airframe object's heading and altitude are controlled. Whereas a real autopilot outputs roll and pitch cues which translate into rates of changes in heading and altitude (2:15), this autopilot object outputs the desired heading and the altitude error. The desired values are determined by adding the actual missile value from the Airframe with the corresponding error value from Guidance.

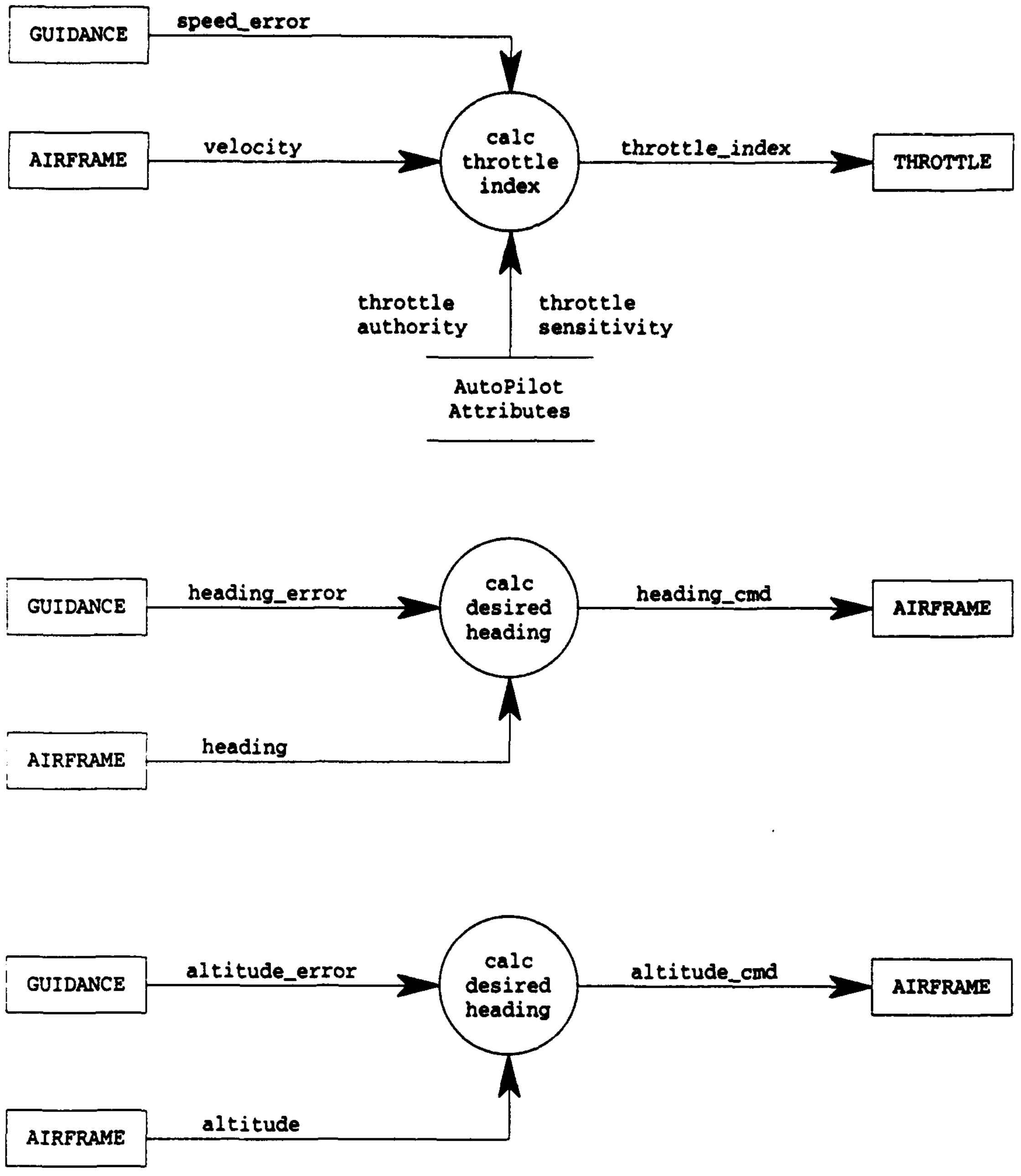


Figure 23. Functional Model for AutoPilot

The throttle-index does not control velocity directly; it controls the Propulsion subsystem's thrust, which alters the Airframe's acceleration. The required throttle setting to correct a given speed error depends upon several parameters, including the Airframe object's current velocity and drag-coefficient, the Throttle object's max-flow-rate and the Jet-Engine object's thrust-factor.

Two simplifying assumptions makes it simple to calculate the desired throttle-index. The first assumption is that drag is simply proportional to the square of the airspeed. In reality, drag has a second component that is inversely proportional to the square of the airspeed (9:138). Assuming high relative airspeeds, the second component can be ignored for the purposes of this research. The second simplifying assumption is that the missile's vertical velocity component can be ignored in calculating the required throttle-index. This assumption would result in higher than desired speeds during descents and slower than desired speeds during climbs. Since cruise missiles usually fly at fairly constant altitudes (13:222), this assumption seems reasonable.

A missile in level flight will have zero acceleration when the thrust and drag have equal magnitudes.

$$Thrust - Drag = 0$$

From the description of the Propulsion subsystem above, thrust was found to be equal to the product of the fuel-consumption-rate and the thrust-factor. Also, the fuel consumption-rate was found to be equal to the product of the throttle-index and the max-flow-rate. Thus,

$$Thrust = ThrottleIndex \times MaxFlowRate \times ThrustFactor$$

By the first simplifying assumption above,

$$Drag = C_d * V^2$$

where C_d is the drag-coefficient and V is the airspeed.

Combining the last two equations and solving for throttle-index yields,

$$ThrottleIndex = \frac{C_d \times V^2}{MaxFlowRate \times ThrustFactor}$$

$$= ThrottleSensitivity \times V^2$$

where all the constant terms have been lumped into a new constant, *Throttle-Sensitivity*. To ensure proper operation, this AutoPilot attribute must be set to the correct value when composing an application. The throttle-authority attribute defines the minimum value that Autopilot can set for the throttle-index (so as not to set the throttle-index below that required to idle the engine).

4.3.7 Warhead Subsystem. The warhead subsystem consists only of the Warhead object. The warhead's function is to terminate the simulation by sending a STOP-EVENT to the application executive event manager when it receives a detonate signal from guidance. Warhead has the inputs, outputs, and internal attributes shown in Table 13.

Table 13. Warhead

Inputs	Outputs	Attributes
current-time arm detonate		yield armed?

Functional flow and state diagrams were not required for the Warhead object due to its simplicity. The warhead has two states: idle and armed. Warhead begins in the idle state. When guidance determines that the next-to-last route point has been reached, it sends an arm signal to the warhead causing the warhead to transition to the armed state. Once in the armed state, warhead waits for guidance to send a detonate signal when guidance determines that the missile has reached the last route point (assumed to be the target point). Upon receipt of the detonate signal, warhead sends a STOP-EVENT to the event manager to terminate the simulation. If the missile fails to reach the target for some reason (runs out of fuel, incorrectly specified parameters, etc.), there should be a backup Stop-Event in the event manager to terminate the simulation.

Figure 24 shows the functional model of the entire cruise missile.

4.4 Stage 3: Define/refine domain-specific design and implementation constraints.

The purpose of this stage of the process is to identify constraints on the architecture imposed by applications in the domain. There are two broad categories of constraints: those that ensure *correctness* of the applications, and those that ensure *efficiency* of the applications.

The architectural modifications made to support the event-driven logic circuits domain were found to be adequate to support correct execution in the time-driven mode. No additional architectural requirements were identified for the cruise missile domain, but new executive services were required to support the time-driven applications.

4.5 Stage 4: Develop domain architectures/models

No additional architectural modifications were identified for this domain; time-driven executive services were developed by Welgan (25).

In determining how to control the missile's speed, attributes in four different primitives were found to be inter-related: Airframe drag-coefficient, the Jet-Engine thrust-factor, Throttle max-flow-rate, and Autopilot throttle-sensitivity. If the application specialist is not aware of the relationships between these attributes, he could specify an erroneous application that does not control the missile's speed as desired. To prevent this from occurring, and to help alleviate the burden on the application specialist, domain-specific semantic checks were developed to verify that the four attributes are compatible prior to execution. If the proper relationships are not met, the application specialist is warned and shown the correct value for throttle-sensitivity. The architectural interface to the domain-specific semantic checks discussed in Section 3.5.4 was followed for this domain.

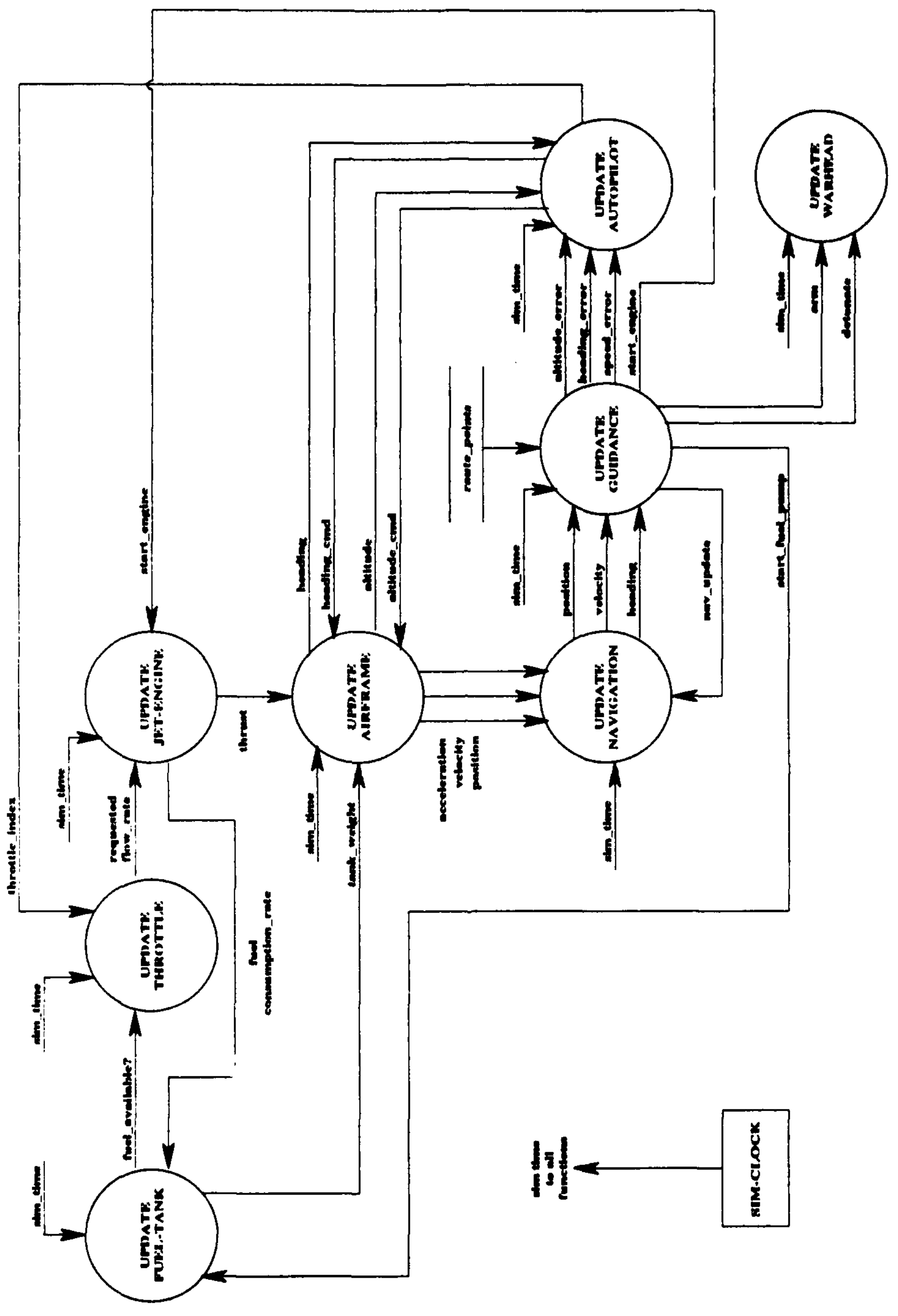


Figure 24. Functional Model of the Cruise Missile

4.6 Stage 5: Produce/gather reusable workproducts

4.6.1 Development. The eight objects identified in stage 2 were mapped into Architect during this stage. There were no existing components, as the circuits domain had, so all objects were developed from scratch. The first step was to develop a Refine domain model that defined the object class hierarchy of the domain objects. This hierarchy paralleled the structure identified in the object model of Figure 16. Also included in the domain model were domain-specific data types required by the primitives. These included three-dimensional vectors used for position, velocity, and acceleration data, and route-point definitions.

The next step was to code the primitives in Refine. The three primitives in the propulsion subsystem were coded first. The propulsion primitives provided a relatively simple set of inter-related primitives that could later be used to test the time-driven executive.

The OCU model was found to be quite satisfactory for this domain. The time-driven primitives exactly paralleled the event-driven primitives in structure. Each primitive has Update and SetState functions, a set of attributes, and uses the same architectural interfaces that the event-driven primitives use. Internally, the only difference between these time-driven primitives and the event-driven primitives is that the time-driven primitives do not raise application events.

In the event-driven domain, primitives did not know about the passage of time except through the receipt of SetState events. The time-driven primitives receive time as an input, so they do not need to raise SetState events to inform them that time has elapsed. Since the time-driven primitives do not generate SetState events, there is no need to generate Remove events.

When an event-driven primitive calls the set-export function to update its external state, the architecture returns a list of Update events for the consumers of that external state as part of the stimulus-response paradigm. The time-driven primitives operate with a different paradigm; they are updated on a periodic schedule by the executive. The stimulus-response paradigm does not apply to them. The Update events returned to the time-driven primitives by the set-export function are filtered out by the primitives and not passed up the subsystem tree to the executive. Thus, the same architectural interface can be used for the event-driven and time-driven primitives.

After the primitives were coded, a domain-specific language was developed using the Dialect grammar tool as discussed in Section 3.6.3. The DSL was used to create applications and parse them into Architect for testing. Validation of the cruise missile technology base is discussed in the following section.

4.6.2 Validation. Testing the cruise missile objects was more difficult than the logic circuits objects because of their increased complexity and the interdependencies among the objects. The propulsion subsystem objects were tested first because of their concurrent development with the executive. The propulsion subsystem requires two external inputs to start the tank's fuel pump and to start the engine. These inputs were specified to be of category *signal* and type *boolean*, the same as the inputs and outputs in the circuits domain. For test purposes, the DSL was modified to include a switch object from the circuits domain so that a switch could be used to supply the needed inputs. The switch was modified so its operation did not require it to raise a SetState event. The thrust and weight outputs from the subsystem were monitored with debug statements in the engine and tank primitives, respectively. Other internal attributes were examined using the Refine *Browse* function to examine the object base.

After the propulsion subsystem and the executive were validated, a complete missile application was composed following the structure shown in Figure 16. The application was defined as a top-level subsystem, representing the missile, with subordinate propulsion, airframe, avionics, and warhead subsystems. Each subordinate subsystem in turn contained the constituent objects making up that subsystem. The domain objects were defined with the following attribute values:

```
fuel-tank the-tank
  capacity: 720.0                 % in mass or volume units
  empty weight: 50.0              % pounds
  fuel level: 100.0               % same units as capacity
  flow rate: 0.0
  pump off
  last time: 0
  fuel density: 1.0               % pounds/unit of capacity

throttle the-throttle
  max flow rate: 0.2              % pounds/sec

jet engine the-engine
  thrust factor: 62500.0          % seconds (specific impulse)
  max flow rate: 0.2              % pounds/sec (recommend same as throttle)
  mode: off

airframe the-airframe
  position: (0.0, 0.0, 100.0)     % feet
  velocity: (200.0, 0.0, 0.0)     % fps
  acceleration: (0.0, 0.0, 0.0)   % fps/sec
  heading: 0.0                    % radians
  turn rate: 0.0                  % radians/sec
  max g: 5.0                      % fps/s ("g"-turn limit)
  lift coef: 0.4                  % (not currently implemented)
  drag coef: 0.05                 % pounds-sec-sec/ft-ft
  weight: 2000.0                  % weight in pounds less tank weight
  last time: 0

navigation the-navigation
  position: (0.0, 0.0, 100.0)     % (same as airframe position)
  velocity: (200.0, 0.0, 0.0)     % (same as airframe velocity)
  acceleration: (0.0, 0.0, 0.0)   % (same as airframe acceleration)
  error factor: 0.00              % (recommend less than 0.10)
  mode: align

guidance the-guidance
  route points: ( 4000.0,    0.0, 100.0) route  20,
                ( 4000.0, 4000.0, 100.0) route  35,
                (    0.0, 4000.0, 100.0) route  50,
                (    0.0,    0.0, 100.0) target 65

autopilot the-autopilot
  throttle sensitivity: 0.000004  % (set by semantic checks)
  throttle authority: 0.05        % (minimum throttle setting)

warhead the-warhead
```

Engineering judgement was used to determine attribute values for the domain objects. These values define a logically consistent composition, but are not intended to represent actual values of any specific missile. As such, this should be regarded as an illustrative example.

The fuel tank initially contains 100 pounds of fuel although its capacity is 720 pounds. The two engine attributes determine its maximum thrust. This engine can provide up to 12,500 pounds of thrust when consuming fuel at the rate of 0.2 pounds/second. The engine's thrust and the airframe's drag coefficient determine the missile's maximum airspeed in level flight.

$$V_{MAX} = \sqrt{Thrust_{MAX}/C_d}$$

In this case, the missile's maximum airspeed is 500 fps (435 knots). The airframe is also specified to weigh 2000 pounds and be capable of making a 5 'g' turn.

The initial conditions for the missile are defined in the airframe object's position, velocity, heading, and turn rate attributes. In this example, the missile begins at 100 feet over the origin with a constant velocity of 200 feet per second (fps) along the X-axis. The intended flight path is defined by the guidance object's route point list. The four route points in the list define a square with sides of 4000 feet. Although obviously not a realistic flight path, this course was chosen for several reasons. The sharp turns tested the missile's ability to make quick heading adjustments and get back on course toward the next route point. The short distances between route points tested the missile's ability to accurately regulate its airspeed to make the route point rendezvous at the required times. The closed course verified proper behavior in all directions of flight.

The application was then parsed into Architect and executed. Figure 25 shows an X-Y plot of the missile's path as it maneuvered around the closed course.

Figure 26 shows a time plot of the missile's airspeed over the closed course. Beginning with an initial airspeed of 200 feet per second (174 knots), the airspeed drops until the engine begins producing thrust. (The way the engine sequences through its modes has

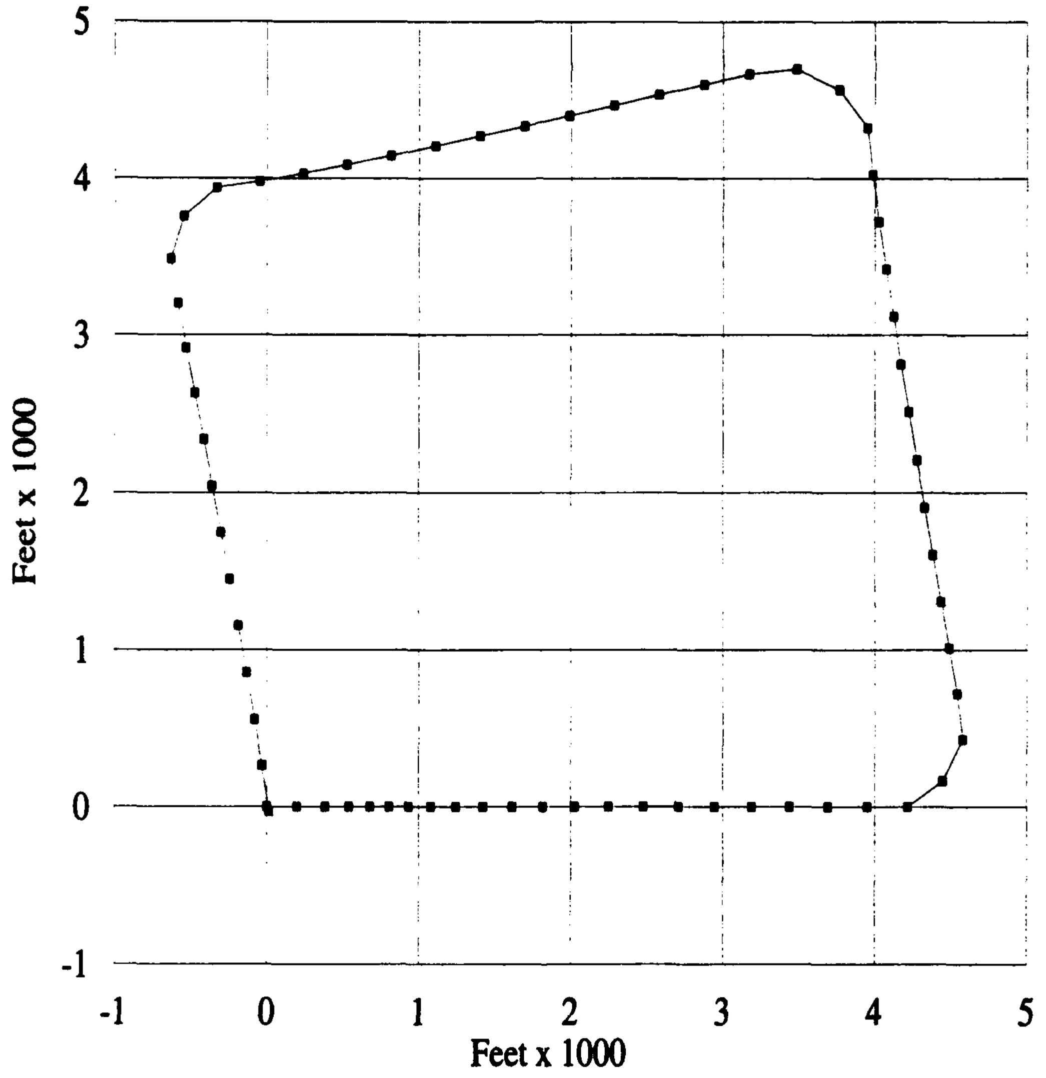


Figure 25. Closed Course Flight Path Performance

since been changed to reduce the long delay time between "launch" and the production of thrust.) The airspeed then increases as the missile seeks to find the right speed to make it arrive at the first route point at time 15. Although the missile was able to meet the first and subsequent route point times, the airspeed curve suggests that a more sophisticated control algorithm might be needed to smooth out the variations during turns.

4.6.3 Performance Considerations. The primitives in this domain are more complex than the circuits domain primitives. They typically have more inputs and outputs and their functions handle arithmetic and trigonometric calculations instead of simple boolean expressions. These primitives can be expected to execute more slowly due to their increased computational requirements. In a time-driven application, the executive will invoke updates on a periodic schedule. The number of update cycles required for an application in this domain will depend upon the simulation time needed to complete the 'mission' and the rate at which updates are invoked.

In a scenario in which the missile flies a distance of 600 nautical miles at an average speed of 300 knots, the application will require two hours of simulated time to execute. If the period of the update schedule is one second, then 7,200 update cycles will occur. Each of the eight primitives is updated during a cycle; therefore, the application will require 57,600 updates. Architect runs in Refine which uses an underlying Lisp environment. Lisp is an interpreted language and executes rather slowly. Long execution times can, therefore, be expected for cruise missile applications. Although execution efficiency is not a goal of Architect, long execution times needed to test and validate a domain can make an environment difficult to work in. Two workarounds can be used to reduce the number of update cycles: the flight time can be reduced and the update period can be lengthened. The first workaround reduces scenario realism; the second reduces simulation accuracy. Fortunately, neither scenario realism nor simulation accuracy were critical to this research. The only available workaround was reduced flight times because the current executive does not support variable update intervals.

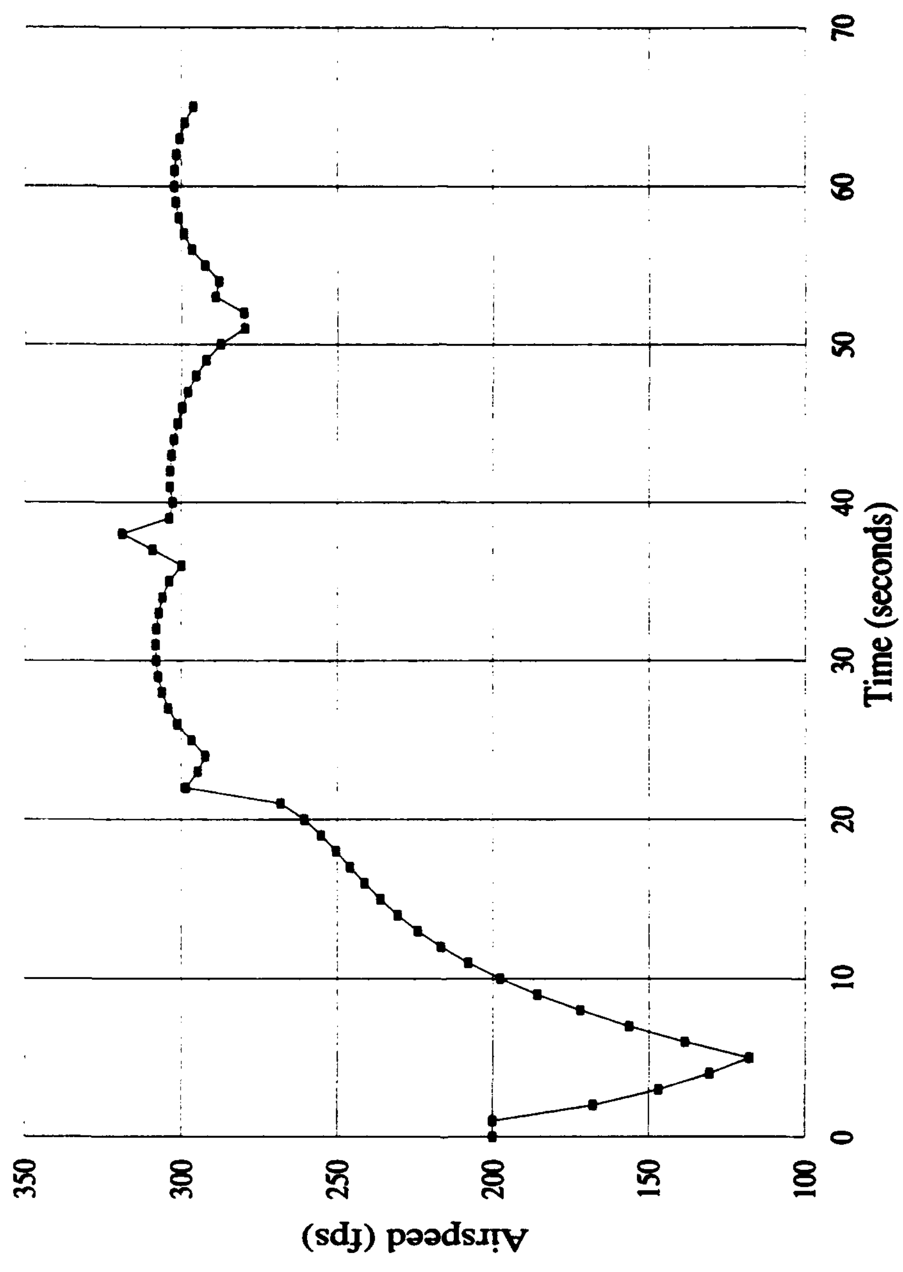


Figure 26. Missile Airspeed

The application composed in Section 4.6.2 required 65 seconds of simulated time to complete. Actual execution time was approximately 25 minutes. Longer 'missions' can be expected to take proportionately longer.

4.7 Summary

Eight time-driven cruise missile domain objects were developed during this part of the research. These objects can be composed into propulsion, avionics, airframe, and warhead subsystems that in turn define a complete cruise missile application. No new architectural modifications to Architect were required to support the time-driven domain. An application was composed and executed to verify proper behavior.

V. Conclusions and Recommendations

5.1 Introduction

This chapter provides a summary of the accomplishments of this research. It begins with a review of the original goals and discusses the results. Then, suggestions for improvements and further research in this area are presented.

5.2 Research Goals

The primary goals of this research were to demonstrate the feasibility of composing time-dependent specifications using the OCU/Architect architecture and to extend the Architect technology base by developing two diverse domain models for time-dependent domains. The first step was to understand the domain analysis process. Domain analysis is a rather new field and its techniques and processes are still maturing. A process recently introduced by Tracz, Coglianese, and Young was chosen as a guide for modeling the two new domains.

5.3 Results

This research showed that the OCU/Architect architecture, with modifications, can support the composition and execution of time-dependent specifications. Demonstration required the simultaneous development of new time-dependent primitives, modification of the Architect architecture, and development of an executive. The new primitives developed during this research drove several changes to Architect's software architecture and executive. These changes were made by Gool (6) and Welgan (25).

5.3.1 Impact on the Architecture and Executive.

- Events were added to support the event-driven and time-driven primitives. The architectural structure was modified to give subsystems 'In-Event' and 'Out-Event' areas.

- Two new architecture functions were developed to provide the interface between the primitives and the executive. These new functions allow the primitives to generate events to control their own behavior.
- The existing *set-export* architecture function was modified to support the stimulus-response paradigm. Whenever an event-driven object updates an export value, the set-export function generates Update-Events for every consumer of that export data.
- Domain knowledge was removed from the subsystem update function. The sequential control algorithm in the subsystem update function was replaced with domain-independent event servicing and routing routines.
- The OCU SetState function was changed from a domain-independent function to a primitive-specific function.
- The ability to preset import and export values prior to execution was incorporated to support applications requiring initial conditions.
- The architectural semantic-checks function now checks for the presence of domain-specific semantic checks to allow for more realistic composition rules.

5.3.2 Architect's New Capabilities. Fourteen event-driven logic circuit domain primitives and eight time-driven cruise missile domain primitives were developed to demonstrate time-dependent behaviors. These new technology bases impart several new capabilities to Architect. Event-driven and time-driven applications can now be composed. These applications exhibit behaviors that could not be specified in Architect prior to this research.

- Delays - Three types of delay were incorporated into the design of the various logic circuits primitives.
    1. Transport Delay - simulates propagation delay through a device, that is, the time delay between when changes on the input are expressed as changes in output.
    2. Inertial Delay - simulates the rejection of "noise" on inputs. Signals must be stable for a period of time before they are considered valid.
    3. Delta Delay - used to enforce 'causality' in the stimulus-response paradigm. Objects without delays cannot update their exports immediately; they must

schedule a set-state event for themselves (with zero time offset) to change their outputs.

- Periodicity - A primitive with an astable output was developed to demonstrate a "free-running" device with periodic behavior.
- Transience - A primitive with a monostable output was developed and tested to demonstrate a device with transient behavior.
- Feedback - Applications with feedback can now be specified if initial conditions are set in the import and export areas.
- Exogenous events - The application specialist can preload events prior to execution. These can be used to initiate execution or change an object's state at predetermined times to control the behavior of the application. They could also be used to insert "malfunctions" into the application.

Despite the enhanced capabilities provided by the event-driven and time-driven technology bases, applications in these domains are easier to compose than the non-event-driven sequential applications because there is no need to specify update algorithms for each subsystem; one or more events can be used to trigger execution of the application. Another product of this research was the incorporation of simple domain-specific semantic checks to provide additional composition rules for circuit domain primitives.

5.4 Evaluation of the Domain Engineering Process

Tracz' five-stage domain engineering process proved to be an acceptable tool for this research. Even though there were differences between what Tracz was trying to achieve and the objectives of this research, the process could be followed with some tailoring.

- *Stage 1: Define the scope of the domain.* For the circuits domain, this stage was not as useful since the research was beginning with an established set of objects. When the need for additional objects was found, this stage was revisited in accordance with the iterative nature of the process. For the cruise missile domain, this stage was directly applicable.
- *Stage 2: Define/refine domain-specific concepts/requirements.* No changes were identified for this stage of the process.
- *Stage 3: Define/refine domain-specific design and implementation constraints.* Only a small part of this stage was applicable. Many of the constraints that need to be

identified are those that affect the real-time performance of the application. Architect is not currently concerned with specifying the real-time behavior of applications. Constraints such as 'how fast', 'how often', and 'how big' are not applicable to Architect. Other constraints that determine the implementation language, accuracy, user interface, etc., do not apply because these are determined by the underlying Refine and Lisp environments upon which Architect runs.

- *Stage 4: Develop domain architectures/models.* Tracz' goal was to develop *domain-specific* software architectures for composing avionics software applications. Architect, in contrast, is designed to be *domain-independent* with respect to its software architecture. The executive is considered part of a DSSA but is a separate entity in Architect. Domain knowledge had to be analyzed for domain-specificity or domain-independence to determine if the knowledge should reside in the domain or in the architecture/executive. When a determination cannot be made due to lack of experience with other domains, the default decision should be to embed the knowledge in the domain. The goal of this stage, then, is to modify the existing architecture and executive to incorporate newly identified *domain-independent* functionality. The domain-specific semantic checks were also developed in this stage.

- *Stage 5: Produce/gather reusable workproducts.* This stage was directly applicable to both domains. No changes to the process were identified.

5.5 Suggestions for Improvements

5.5.1 Event Ordering. Even though causality is enforced by the stimulus-response paradigm at the primitive level, the possibility exists for events to be processed out of order. The architecture handles events in *sets* rather than *sequences*. Since order is not important in sets, there is no guarantee that order will be preserved. Events are sorted at the executive level by time and priority. Events of the same type have equal priority. In a system without delays, all events will have the same time stamp and so many different permutations of order are possible, only one of which may be correct.

5.5.2 Architectural Syntax and Semantic Checks. The architectural syntax and semantic checks enforced by Architect to determine if two components can be connected were suitable for the circuits domain but restrictive for the cruise missile domain.

Import and export objects have two attributes *category* and *type*. For a connection to be made between and import object and an export object, the corresponding values of these two attributes must match. In the circuits domain, all imports and exports are of category *signal* and type *boolean*, allowing the output of any object to be connected to the input of any other object.

In the cruise missile domain, a properly composed application can only be connected in one way. In assigning categories to the different import and export data, the convention of using standard domain terminology such as position, velocity, acceleration, angle, flow-rate, etc., was adopted in the belief that the most general name would provide the greatest flexibility for reuse. While this may be true, it also makes it more difficult for the application specialist to compose applications because he must decide which import or export to select when more than one possibility exists. More sophisticated syntax and semantic rules could resolve the ambiguities and alleviate the burden on the application specialist without compromising reusability.

5.5.3 Visualization Support. Monitoring the execution of an application such as the cruise missile is difficult without graphical support. The Automated Programming Technologies for Avionics Software (APTAS) system has the capability to display motion data from a file (10). Using the display capabilities of APTAS to visualize the results from an execution in Architect would improve the ability to evaluate the performance of cruise missile applications.

5.6 Suggestions for Further Research

Much has been learned about the requirements for a system such as Architect from the single domain of sequentially executed logic circuits. More was learned during this research effort which produced an event-driven logic circuits domain and a time-driven cruise missile domain. But much remains to be learned; following are suggested areas for further research.

- *Extending the Technology Base.* Further study of the new technology bases is needed to improve understanding of the requirements for a composition tool such as Architect. These domains should not be considered definitive, however. Several more technology bases need to be developed to obtain experience with a wide spectrum of domains and execution modes. A moving vehicle technology base executing in the event-driven mode should be developed. Current cruise missile applications require all components to be time-driven at the same rate. Variable update rates and mixed-mode execution with a combination of event-driven and time-driven execution need to be investigated.
- *Representation of Domain Knowledge.* More research needs to be done on encoding domain knowledge not contained within the primitives. Currently only simple domain-specific semantic checks are supported. The cruise missile domain analysis process identified the need for more sophisticated tools to help the application specialist configure the domain objects.
- *Dynamic Domains.* Domains that require dynamic allocation and deallocation of objects require sophisticated domain knowledge above the primitive level. The domain knowledge must determine where a newly created object fits in the application structure, how the object is "connected" to other objects in the application, and, in the sequential non-event driven mode, how to modify the update algorithm of the object's parent subsystem.
- *Multiple-domain Applications.* A domain engineer defines the boundary of a domain by examining the range of applications that are required to be composed; if a new application is later identified, the domain engineer may need to expand the boundary by adding new primitives. If the required primitives already exist in another domain, the domain engineer should not have to recreate them in his domain. Rather, he should be able to pull primitives from other domains into his application. Architect's domain-independent software architecture supports this concept; Dialect's

limit on grammar inheritance does not. When the object-base being developed by Cecil and Fullenkamp (3) comes online, Dialect will no longer be required. Multiple domain applications will then be easy to compose from an architectural (syntax and semantics) viewpoint. Further work is needed to understand how domain-specific knowledge from the different domains interact in a single application.

5.7 Conclusion

This research has given Architect two new validated technology bases which can be used to compose a wide variety of applications. The knowledge gathered in the process has provided a significant step toward better understanding the requirements for a domain-oriented application composition system such as Architect.

Appendix A. REFINE *Code Listings for Architect*

The REFINE source code for Architect and the implemented time-dependent domains may be obtained, upon request, from:

Maj Paul Bailor
AFIT/ENG
2950 P Street
Wright-Patterson AFB, OH 45433-7765

(513)255-9263
DSN 785-9263
email: pbailor@afit.af.mil

Vita

Captain Robert Waggoner was born August 18, 1953 in Washington, Indiana and graduated from Cumberland Valley High School in Mechanicsburg, Pennsylvania in 1971. He enlisted in the United States Air Force in October 1971 and completed technical training for F-4E weapon control systems maintenance at Lowry AFB, Colorado in July 1972. During the next three years, he was assigned to Seymour Johnson AFB, North Carolina; Ubon RTAFB, Thailand; Udorn RTAFB, Thailand; and Eglin AFB, Florida. SSgt Waggoner separated from the Air Force in October 1975. In 1981 he graduated from the Pennsylvania State University at Middletown with a Bachelor of Science degree in Electrical Engineering Technology. He entered Officer Training School at Lackland AFB, Texas in May 1982. After receiving his commission in August 1982, he was assigned to the Directorate of Flight Test Engineering at the 4950th Test Wing at Wright-Patterson AFB, Ohio. For the next four years he directed flight tests for a wide variety of projects including the satellite communications and ionospheric research testbed aircraft. In October 1986 he was reassigned to the Ballistic Missile Office at Norton AFB, California where he managed the requirements analysis and integration for the Small ICBM program. In 1992 he entered the Air Force Institute of Technology to pursue a Master of Science degree in Computer Science.

Permanent address: 314 E. Elnora St
Odon, Indiana 47562

Bibliography

1. Anderson, Cynthia. *Creating and Manipulating Formalized Software Architectures to Support a Domain-Oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/92D-01, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.

2. Batory, Don and Steve Schafer. "A Domain Model for Avionics Software." *Collected Papers of the Domain-Specific Software Architectures (DSSA) Avionics Domain Application Generation Environment (ADAGE)* IBM Federal Sector Company, May 1993.

3. Cecil, Danny A. and Joseph Fullenkamp. *Using Object-Oriented Database Management System (OODBMS) Technologies to Archive Formally Specified Software Artifacts (Draft).* MS thesis, AFIT/GCS/ENG/93D-03, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

4. Cossentine, Jay A. *Developing a Sophisticated User Interface to Support Domain-Oriented Application Composition and Generation Systems.* MS thesis, AFIT/GCS/ENG/93D-04, Graduate School of Engineering, Air Force Institute of Technology (AU), Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

5. D'Ippolito, Richard S. *Using Models in Software Engineering.* Technical Report, Software Engineering Institute, Carnegie Mellon University, 1989.

6. Gool, Warren E. *Alternative Architectures for Domain-Oriented Application Composition and Generation Systems.* MS thesis, AFIT/GCS/ENG/93D-11, Graduate School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1993.

7. Graybeal, Wayne J. and Udo W. Pooch. *Simulation: Principles and Methods.* Winthrop Publishers, Inc., 1980.

8. Holt, Charles A. *Electronic Circuits Digital and Analog.* John Wiley & Sons, Inc., 1978.

9. Houghton, E. L. and A. E. Brock. *Aerodynamics for Engineering Students.* Butler and Tanner, LTD, 1970.

10. Jensen, Paul S. and Lori Ogata. *DRAFT Final Report for Automatic Programming Technologies for Avionics Software (APTAS).* Technical Report, Lockheed Software Technology Center, July 1991.

11. Lee, Kenneth J. and others. *Model-Based Software Development.* Technical Report, Software Engineering Institute, January 1992.

12. Lennox, Duncan, editor. *Jane's Strategic Weapon Systems.* Janes Information Group, Ltd., 1992.

13. Lin, Ching-Fang. *Modern Navigation, Guidance, and Control Processing.* Prentice-Hall, 1991.

14. Lipsett, Roger and others. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 1989.

15. Millman, Jacob. *Microelectronics Digital and Analog Circuits and Systems*. McGraw-Hill Book Company, 1979.

16. Prieto-Díaz, Rubén. "Domain Analysis: An Introduction," *ACM SIGSOFT Software Engineering Notes*, *15*:47–54 (April 1990).

17. Prieto-Díaz, Rubén. "Domain Analysis for Reusability." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 63–69, IEEE Computer Society Press, 1991.

18. Randour, Mary Anne. *Creating and Manipulating a Domain-Specific Formal Object Base to Support a Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/92D-13, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.

19. Reasoning Systems, Inc. DIALECT^{TM} *User's Guide*. Palo Alto, CA, July 1990. For DIALECT^{TM} Version 1.0.

20. Reasoning Systems Inc. REFINE^{TM} *User's Guide*. Palo Alto, CA, 1990. For REFINE^{TM} Version 3.0.

21. Rumbaugh, James and others. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

22. Savant, C. J. Jr and others. *Principles of Inertial Navigation*. McGraw-Hill, Inc., 1961.

23. Tracz, Will etal. "A Domain-Specific Software Architecture Engineering Process Outline." *Collected Papers of the Domain-Specific Software Architectures (DSSA) Avionics Domain Application Generation Environment (ADAGE)* IBM Federal Sector Company, May 1993.

24. Weide, Timothy. *Development of a Visual System Interface to Support a Domain-oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93M-05, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, March 1993.

25. Welgan, Robert L. *Domain-Analysis and Modeling of a Model-Based Software Executive*. MS thesis, AFIT/GCS/ENG/93D-25, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1993	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
DOMAIN MODELING OF TIME-DEPENDENT SYSTEMS

5. FUNDING NUMBERS

6. AUTHOR(S)
Robert W. Waggoner

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/93D-23

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Capt Rick Painter
2241 Avionics Circle, Suite 16
WL/AAWA-1 BLD 620
Wright-Patterson AFB, OH 45433-7765

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research investigated the feasibility of composing time-dependent specifications in Architect, a domain-oriented application composition and generation system being developed at the Air Force Institute of Technology (AFIT). Architect composes formally specified domain objects into an executable software specification that can be used to verify program correctness prior to generation of language specific code. As part of this research, domain modeling techniques were investigated and a candidate process was selected for evaluation. The process was used to develop domain models for two diverse time-dependent domains. Using object-oriented analysis, formal specifications were developed for a collection of event-driven logic circuit components and a collection of time-driven cruise missile components. Applications from each domain were composed in Architect and executed to verify correct behavior.

14. SUBJECT TERMS
software engineering, domain modeling, domain analysis, time-dependent systems, knowledge based systems, application composition systems

15. NUMBER OF PAGES
89

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL